Developing High Performance Applications



Java NIO

Ron Hitchens 著

裴小星 译

O'REILLY®

VISIT…

LANZAROTE Caliente.COM

Java™ NIO

Ron Hitchens 著

裴小星 译

出版商：O’Reilly

第一版 2002 年 8 月

ISBN: 0-596-00288-2

Java NIO 深入探讨了 1.4 版的 I/O 新特性，并告诉您如何使用这些特性来极大地提升您所写的 Java 代码的执行效率。这本小册子就程序员所面临的有代表性的 I/O 问题作了详尽阐述，并讲解了如何才能充分利用新的 I/O 特性所提供的各种潜能。您将通过实例学会如何使用这些工具来解决现实工作中常常遇到的 I/O 问题，并了解这些新特性如何对响应速率、可伸缩性和可靠性产生直接影响。

NIO API 是对 1.3 版 I/O 特性的补充而非取代，因此，何时使用新的 API，何时老的 1.3 版 I/O API 更适合特定应用，也是您将学习的内容。

目录

题献 **1**
前言 **2**
组织形式 3
目标读者 5
软件及版本 5
本书中使用的约定 6
如何联系我们 7
鸣谢 8
第一章　简介 **10**
1.1 I/O 与 CPU 时间的比较 10
1.2 CPU 已不再是束缚 11
1.3 进入正题 12
1.4 I/O 概念 13
1.5 总结 20
第二章　缓冲区 **22**
2.1 缓冲区基础 23
2.2 创建缓冲区 36
2.3 复制缓冲区 38
2.4 字节缓冲区 40
2.5 总结 51
第三章　通道 **53**
3.1 通道基础 55
3.2 Scatter / Gather 62
3.3 文件通道 68
3.4 内存映射文件 83
3.5 套接字通道 96
3.6 管道 117
3.7 通道实用工具类 122
3.8 总结 123
第四章　选择器 **125**
4.1 选择器基础 126
4.2 使用选择键 132
4.3 使用选择器 135
4.4 异步可关闭性 144
4.5 选择缩放 144
4.6 总结 148

第五章　正则表达式......150
5.1 正则表达式基础......151
5.2 Java 正则表达式 API......153
5.3 字符串类正则表达式方法......153
5.4 Java 正则表达式语法......174
5.5 面向对象的文件检索......180
5.6 总结......186
第六章　字符集......187
6.1 字符集基础......187
6.2 字符集......188
6.3 字符集服务提供接口......207
6.4 总结......217
附录 A. NIO 与 JNI......218
附录 B. 可选择通道 SPI......220
附录 C. NIO 快速参考......223
C.1 java.nio 包......223
C.2 java.nio.channels 包......230
C.3 java.nio.channels.spi 包......242
C.4 java.nio.charset 包......244
C.5 java.nio.charset.spi 包......248
C.6 java.util.regex 包......248

题献

给我的妻子，Karen。

离了你我可怎么办？

前言

计算机毫无用处，除了答案什么也没有。

——毕加索

本书介绍了 Java 平台上的高级输入 / 输出，具体点说，就是使用 Java 2 标准版（J2SE）软件开发包（SDK）1.4 及以后版本进行的输入 / 输出。J2SE 1.4 版代号 Merlin，包含可观的 I/O 新特性，对此我们将作详细论述。这些新的 I/O 特性主要包含在 `java.nio` 软件包及其子包中，并被命名为 New I/O（NIO）。通过本书，您将学会如何使用这些令人兴奋的新特性来极大地提升 Java 应用程序的 I/O 效率。

Java 真正的归宿还在企业应用（何谓“企业应用”，恐怕还没有一个确切的定义），但与本地编译语言相比，Java 在 I/O 领域一直处于劣势，这种情况直到 J2SE SDK 发布了 1.4 版以后才有了改观。Java 的劣势源于其最大的优势：一次编写，到处运行。Java 需要运行于虚拟机（即 JVM）之上，为了保证 Java 字节码在各种 JVM 部署平台上运行效果一致，作些妥协是必须的。既然需要通用于不同的操作系统平台，那么，某种程度上就必须选择各种平台都接受的处理方案。

最切实地感受到妥协带来的后果的，莫过于 I/O 领域。虽然 Java 有一套完备的 I/O 类，但迄今为止还只是针对通用特性，通常位于高端抽象层，横跨各种操作系统。这些 I/O 类主要面向流数据，经常为了处理个别字节或字符，就要执行好几个对象层的方法调用。

这种面向对象的处理方法，将不同的 I/O 对象组合到一起，提供了高度的灵活性，但需要处理大量数据时，却可能对执行效率造成致命伤害。I/O 的终极目标是效率，而高效的 I/O 往往又无法与对象形成一一对应的关系。高效的 I/O 往往意味着您要选择从 A 到 B 的最短路径，而执行大量 I/O 操作时，复杂性毁了执行效率。

传统 Java 平台上的 I/O 抽象工作良好，适应用途广泛。但是当移动大量数据时，这些 I/O 类可伸缩性不强，也没有提供当今大多数操作系统普遍具备的常用 I/O 功能，如文件锁定、非块 I/O、就绪性选择和内存映射。这些特性对实现可伸缩性是至关重要的，对保持与非 Java 应用程序的正常交互也可以说是必不可少的，尤其是在企业应用层面，而传统的 Java I/O 机制却没有模拟这些通用 I/O 服务。

具体的企业在具体的系统上配置具体的应用，这里无关乎抽象。在现实世界，效率是大事——头等大事。企业购置的用于部署大型应用的计算机系统，其 I/O 性能异常卓越（系统供应商往往投入巨资进行研发），而 Java 迄今为止一直无法充分利用这一点。当企业的需求是以最快的速度传送大量数据时，样貌朴实但迅捷的解决方案往往胜过漂亮却动作迟缓的。总之一句话，时间就是金钱。

JDK 1.4 是由 Java 社区进程（Java Community Process）主导发行的首个主要发行版。Java 产品的用户和供应商可就 Java 平台需要引入什么新特性提出要求和建议，JCP（http://jcp.org/）为此提供了手段。本书的主题——Java New I/O（NIO）——就是这样一项提议的产物。Java 规范请求#51（JSR 51, http://jcp.org/jsr/detail/51.jsp）包含了对高速、可伸缩 I/O 特性的详尽描述，借助这一特性，底层操作系统的 I/O 性能可以得到更好发挥。JSR 51 的实现，其结果就是新增类组合到一起，构成了 `java.nio` 及其子包，以及 `java.util.regex` 软件包，同时现存软件包也相应作了几处修改。JCP 网站详细介绍了 JSR 的运作流程，以及 NIO 从最初的提议到最终实现并发布的演进历程。

随着 Merlin 的发布，操作系统强大的 I/O 特性终于可以借助 Java 提供的工具得到充分发挥。论及 I/O 性能，Java 再也不逊于任何一款编程语言。

组织形式

本书分六章，每章针对 NIO 的一个大的方面。第一章讨论一般 I/O 概念，为以后各章相关论述作了铺垫。第二至第四章论及 NIO 的核心内容：缓冲区、通道和选择器。接下来介绍新引入的正则表达式 API。正则表达式的处理与 I/O 紧密贴合，亦被纳入 JSR 51 特征集之内。最后，我们看一下新的可插拔字符集映射系统，这也是 NIO 和 JSR 51 的组成部分。

以下概要，是专门为那些迫不及待往前赶的人准备的。

缓冲区（Buffers）

新的 *Buffer* 类是常规 Java 类和通道之间的纽带。原始数据元素组成的固定长度数组，封装在包含状态信息的对象中，存入缓冲区。缓冲区提供了一个会合点：通道既可提取放在缓冲区中的数据（写），也可向缓冲区存入数据供读取（读）。此外，还有一种特殊类型的缓冲区，用于内存映射文件。

第二章将详细讨论缓冲区对象。

通道（Channels）

NIO 新引入的最重要的抽象是通道的概念。*Channel* 对象模拟了通信连接，管道既可以是单向的（进或出），也可以是双向的（进和出）。可以把通道想象成连接缓冲区和 I/O 服务的捷径。

某些情况下，软件包中的旧类可利用通道。为了能够向与文件或套接字关联的通道进行存取，适当的地方都增加了新方法。

多数通道可工作在非块模式下，这意味着更好的可伸缩性，尤其是与选择器一同使用的时候。

通道会在第三章作详细介绍。

文件锁定和内存映射文件（File locking and memory-mapped files）

新的 *FileChannel* 对象包含在 `java.nio.channels` 软件包内，提供许多面向文件的新特性，其中最有趣的两个是文件锁定和内存映射文件。

在多个进程协同工作的情况下，要协调各个进程对共享数据的访问，文件锁定是必不可少的工具。

将文件映射到内存，这样在您看来，磁盘上的文件数据就像是在内存中一样。这利用了操作系统的虚拟内存功能，无需在内存中实际保留一份文件的拷贝，就可实现文件内容的动态高速缓存。

文件锁定和内存映射文件也在第三章讨论。

套接字（Sockets）

套接字通道类为使用网络套接字实现交互提供了新方法。套接字通道可工作于非块模式，并可与选择器一同使用。因此，多个套接字可实现多路传输，管理效率也比 `java.net` 提供的传统套接字更高。

三个新套接字通道，即 *ServerSocketChannel*、*SocketChannel* 和 *DatagramChannel*，将在第三章讲到。

选择器（Selectors）

选择器可实现就绪性选择。*Selector* 类提供了确定一或多个通道当前状态的机制。使用选择器，借助单一线程，就可对数量庞大的活动 I/O 通道实施监控和维护。

选择器会在第四章作详细介绍。

正则表达式（Regular expressions）

新增的 `java.util.regex` 软件包将类似 Perl 语言的正则表达式处理机制引入 Java。这一人们期盼已久的特性有着广泛用途。

新的正则表达式 API 之所以被看成是 NIO 的组成部分，是因 JSR 51 把它与其他 NIO 特性放在一起作了详细说明。虽然它在许多方面与 NIO 的其他组成部分缺乏平行关系，但它在文件处理等众多领域都是极其有用的。

第五章讨论 JDK 1.4 正则表达式 API。

字符集（Character sets）

`java.nio.charsets` 提供了新类用于处理字符与字节流之间的映射关系。您可以对字符转

换映射方式进行选择，也可以自己创建映射。

字符代码转换的相关问题在第六章讨论。

目标读者

本书为中高级 Java 程序员所写：他们熟练掌握这门语言，在完成大规模、复杂的数据处理任务时，有充分利用 Java NIO 所提供之新特性的愿望和需求。在写作的过程中，我假定您对 JDK 标准类软件包、面向对象的设计技巧、继承等等都有充分了解。我还假定您了解 I/O 在操作系统层面的基本工作原理，知道什么是文件，什么是套接字，什么是虚拟内存，诸如此类。第一章就这些概念作了高屋建瓴的回顾，但没有深度剖析。

如果您对 Java 平台上的 I/O 软件包尚处于摸索阶段，也许您应该先读一读 Elliote Rusty Harold 所著《Java I/O》（O'Reilly）（http://www.oreilly.com/catalog/javaio/）。那是一本关于 `java.io` 软件包的很好的入门教材。虽然可以把本书看作那本书的后续读物，但两者并无承接关系。本书重点关注的是如何利用新的 `java.nio` 软件包实现 I/O 性能的最大化。另外，本书还引入了一些新的 I/O 概念，这也超出了 `java.io` 软件包的范畴。

我们还探讨了字符集编码和正则表达式，这也是与 NIO 绑定在一起的新特征集的一部分。为软件国际化或专业应用使用字符集的程序员会对 `java.nio.charsets` 软件包感兴趣，第六章会有相关讨论。

如果您已转向 Java，但时不时地需要回到 Perl 处理正则表达式，告诉您，以后再也不用了。新推出的 `java.util.regex` 软件包在标准 JDK 中包含了 Perl 5 所有的正则表达式功能，个别极其晦涩难懂的除外，此外还有几项创新。

软件及版本

本书描述了 Java 的 I/O 性能，尤其是 `java.nio` 和 `java.util.regex` 两个软件包。这两个软件包首次出现于 J2SE 1.4 版，所以，您必须拥有 Java SDK 1.4（或以后）的可用版本，才能使用本书提供的素材。您可通过访问 Sun 公司网站 http://java.sun.com/j2se/1.4/获得 Java SDK。我还会在正文中使用 J2SE SDK 指代 Java 开发包（JDK），在本书范围内，二者含义相同。

本书基于 2002 年 2 月发布的 SDK 最终版本 1.4.0。早前几个月，1.4 beta 版已广为流传。在最终版本即将发布之前，NIO API 作了重要修正。因此，您会发现本书所讲某些细节，与您在最终版本发布之前所知论述存在出入。本书根据所有已知的最后修正作了更新，应该不存在与最终版本 1.4.0 不一致之处。J2SE 的后续版本可能引入与本书相冲突的内容，如果存在疑问，请参考与软件一同发布的技术文档。

本书包含众多实例，向您示范如何使用 API。所有代码举例及相关信息都可从 http://www.javanio.info/下载，更多实例和测试代码也可从该网站取得。NIO 执行小组提供的额外代码示例可从 http://java.sun.com/j2se/1.4/docs/guide/nio/example/取得。

本书中使用的约定

像所有程序员一样，我对代码的排版样式也有宗教般的虔诚。本书所举范例系根据我个人喜好进行的排版，在相当程度上也符合惯例。我信奉的标准是，一个制表符缩进八格，外加大量的空白分隔符。有些代码举例迫于空间有限，把缩进减少到四格。网站上供下载的源代码使用制表符产生缩进。

当我就 API 举例，展示 JDK 中某个类的方法列表时，我一般只列出上下文提到的特定方法，而把无关方法略去。我通常在章节的开头提供完整的 API 列表，而在随后的具体讨论环节列出其子集。

这些 API 示例通常在语法上并不正确，只是若干方法签名的片段，没有方法主体，仅用于列举可用方法及认可参数。例如：

```
public class Foo
{
        public static final int MODE_ABC
        public static final int MODE_XYZ
        public abstract void baz (Blather blather);
        public int blah (Bar bar, Bop bop)
}
```

本例中，方法 *baz( )*在语法上是完整的，因为抽象声明只包含签名。但 *blah( )*缺少分号，暗示在该类定义中应当有方法主体跟随其后。当我罗列公共域定义常数，如 `MODE_ABC` 和 `MODE_XYZ` 的时候，有意省略了初始值，因为省去的信息不重要。既然定义了公共名，就可以在不知道值的情况下使用该常数。

只要有可能，我尽量直接从随 1.4 JDK 分发的代码截取 API 信息。我着手写作本书的时候，JDK 的版本是 1.4 beta 2。为了保证代码片段是最新的，我尽了一切努力，如有疏漏，在此表示歉意，JDK 提供的源代码才是最终权威。

字体约定

本书采用 O'Reilly 标准字体约定，这并非完全出于个人选择。本书手稿直接以 XML 格式创作，使用纯 Java GUI 编辑器（XXE，http://www.xmlmind.com/）。该软件强制使用 DTD I 及 O'Reilly 的 DocBook 子集，因此，我从未指定字体或字形。我可以选择 XML 元素，如

`<filename>`或`<programlisting>`，然后 O'Reilly 的排版软件会采用合适的字形。

读者当然不必关心这些。那么，以下就是本书所用字体约定的简要说明：

Italic（斜体）用于：

- 路径名、文件名和程序名
- 互联网地址，如域名、URL
- 新术语及其定义

`Constant Width`（等宽字体）用于：

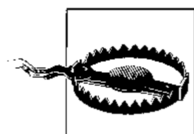
- Java 代码中的名称和关键字，包括方法名、变量名和类名
- 程序清单和代码片段
- 常数值

`Constant Width Bold`（等宽粗体）用于：

- 强调代码示例的重点部分



这个图标表示提示、建议或概括性的说明。



这个图标表示一个警告或者注意事项提醒。

如何联系我们

我虽不是头回写书，却是头回为一般出版物而写。写书可比读书难多了，要解释清楚和 Java 有关的命题，尤其让人心生恐惧，因为 Java 涉及面广，发展迅速，还有为数众多聪明绝顶的人士，芝麻大点的错误也能指出来。

任何意见，不管是正面的还是负面的，我都愿意倾听。我自信在这个项目上做足了功课，但错误在所难免。有关本书组织结构和内容安排的建设性意见，我尤其感兴趣。我力求本书条理清晰、布局合理，也力求有完备的交叉引用，为在各章节随意跳转提供便利。

报酬丰厚的顾问合同、演讲邀约以及任何免费资料，我都欢迎。无聊邮件、垃圾邮件，我将一笑置之。

通过邮箱 ron@javanio.info，或者登陆 http://www.javanio.info/，都可与我取得联系。

O'Reilly 及我本人已竭尽所能对书中信息进行了核实，但某些特性可能已发生改变（抑或我们确有谬误！），不管各位发现什么错误，或者对未来版本有任何建议，都请一并告知。信可写到：

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938（美国及加拿大）
(707) 829-0515（国际／当地）
(707) 829-0104（传真）

您还可通过电邮与 O'Reilly 取得联系。订阅邮件或索取目录，发个消息到：

info@oreilly.com

我们为本书设立了网页，勘误表、示例及任何补充信息都在上面。网址为：

http://www.oreilly.com/catalog/javanio/

询问技术问题，或发表评论，发邮件到：

bookquestions@oreilly.com

欲了解与 O'Reilly 图书、会议、资源中心及 O'Reilly 网络有关的更多信息，留意 O'Reilly 网站：

http://www.oreilly.com/

鸣谢

攒一本书出来可不轻松，哪怕是像本书这样的中小部头。因此，我要向在本书写作过程中给予我帮助的几位朋友表示感谢。

首先，我要感谢本书编辑 Mike Loukides，为我提供了加入 O'Reilly 作家队伍的机会。我是如何赢得 O'Reilly 的出书合同的，至今都觉得不可思议。那是一份荣耀，也是一份责任。谢谢，Mike，逗号用得不对，还请见谅。

我还要感谢 Bob Eckstein 和 Kyle Hart（也是 O'Reilly 的），感谢 Bob 在本书最初的几稿给予我的帮助，感谢 Kyle 给我提供了 JavaOne 年会的免费资料（对了，那次市场营销没准也帮助不小）。Jessamyn Read 把我的涂鸦变成了专业图表。我还要感谢多产的 David Flanagan，在他的《Java 技术手册》（第四版）（*Java in a Nutshell, Fourth Edition* [O'Reilly]）中提及了我微不足道的贡献，并允许我使用该书中的正则表达式语法表。

写技术书籍的极为倚重专家的意见，那是他们发现错误、排堵漏洞的保障。如果素材很新又快速演进，技术上的把关就显得尤为重要，NIO 即属于这种情况。我着手此项目之初，1.4 API 不夸张地讲就是个移动的靶子。万分幸运的是，Sun Microsystems 的 Mark Reinhold 同意为我审稿。他主导着 JSR 51 的技术规范，同时还负责编写 JDK 1.4 中大量的 NIO 代码。在本书初具雏形、还很不完善的时候，Mark 就已参与审稿。他不厌其烦地向我解释诸多疑点，还提出了颇具洞察力的建议，对我帮助极大。在 1.4.1 版即将成形之际，他还停下手中的工作，专门就本书最终稿提出详尽意见。谢谢，Mark。

参与审稿并提出中肯意见的还有另几位仁兄。Jason Hunter（http://www.servlets.com/）仅用了几个小时就如饥似渴地吞下了第一稿，还对章节的组织方式提出了宝贵意见。Digital Gamers, Inc.的 John G. Miller, Jr.（johnmiller@digigamers.com, http://www.digigamers.com/）可谓一丝不苟，他把草稿中的代码示例逐行过了一遍。John 将 NIO 大规模地用于在线互动游戏环境，他丰富的实际工作经验令本书的面貌大为改观。Will Crawford（http://www.williamcrawford.info/）百忙之中挤出时间读完了全部手稿，他的反馈意见可谓一针见血、切中要害。

我还想感谢Keith J. Koski和Michael Daudel（mgd@ronsoft.com）。我们同属一个由Unix和Java高手组成的快乐团队，团队成员在一起共事多年，人称Fatboys。如今Fatboys陆陆续续地瘦了，结婚了，搬到了郊区，有了孩子（包括我本人在内），但只要Bill还用吸管喝肉汤，Fatboy的梦就会继续做下去。Keith和Mike读了最初的几稿，测试了代码，提了建议，给了鼓励。真是无所不能啊 *，哥儿们，多谢了！

最后，我还要感谢我的妻子 Karen。她对技术的东西并不在行，但她聪明、体贴，她的爱就是我的精神食粮。她点亮我的灵魂，赋予我理性，我们在一起共谱生活篇章。

* 译注：此处原文为“phaser enriched”。Phaser 为《星际迷航》（*Star Trek*）中的一种激光武器，功能繁多，适应性极广。作者为了表达对朋友的感激之情，将朋友比作增强版的 phaser。有关 phaser 的更多信息，参见 http://en.wikipedia.org/wiki/List_of_weapons_in_Star_Trek#Phasers。

第一章　简介

先把事实搞清楚，歪曲是以后的事。

——马克·吐温

我们谈谈 I/O 吧。别走哇，回来！I/O 其实没那么枯燥。输入 / 输出问题（I/O）虽谈不上多吸引人，却很重要。程序员多半把 I/O 等同于疏通下水管道：无疑很重要，没有不行，但要是直接跟它打交道，就没那么惬意了，搞不好弄得浑身臭哄哄的。本书要讲的可不是管道疏通，但是阅读了随后章节，您就会知道如何让您的数据流动得稍微顺畅一些。

面向对象的程序设计讲的无非就是封装。封装是个好东西：它分解任务，隐藏实施细节，提高对象的重复利用率。这样的分解、整合既适用于程序，也适用于程序员。您没准就是一位技艺高超的 Java 程序员，创建极其复杂的对象，完成惊世骇俗的任务，而对支撑 Java 平台的基本 I/O 概念却几乎一无所知。本章，我们暂且把封装问题抛在一边，先来看看某些底层 I/O 实施细节，希望有助于您更好地组织协调各个零部件的 I/O 操作。

1.1 I/O与CPU时间的比较

程序员多半当自个儿是软件大师，设计出精巧的例程，这儿压缩几个字节，那儿解开一个循环，要不就在别处作些调整，让对象更加牢固。这些事情当然很重要，乐趣也不少，但是代码优化所带来的回报，可能轻易就被低效的 I/O 所抵销。I/O 操作比在内存中进行数据处理任务所需时间更长，差别要以数量级计。许多程序员一门心思扑在他们的对象如何加工数据上，对影响数据取得和存储的环境问题却不屑一顾。

表 1-1 所示为对数据单元进行磁盘读写所需时间的假设值。第一列为处理一个数据单元所需平均时间，第二列为对该数据单元进行磁盘读写所需时间，第三列为每秒所能处理的数据单元数，第四列为改变第一第二列的值所能产生的数据吞吐率的提升值。

表 1-1. 处理时间与 I/O 时间对吞吐率的影响比较

处理时间（ms）	**I/O 时间（ms）**	**吞吐率（units/sec）**	**增益（%）**
5	100	9.52	（基准）
2.5	100	9.76	2.44
1	100	9.9	3.96
5	90	10.53	10.53
5	75	12.5	31.25
5	50	18.18	90.91
5	20	40	320
5	10	66.67	600

前三行显示了处理阶段的效率提升会如何影响吞吐率。把单位处理时间减半，仅能提高吞吐率 2.2%。而另一方面，仅仅缩短 I/O 延迟 10%，就可使吞吐率增加 9.7%；把 I/O 时间减半，吞吐率几乎翻番。当您了解到 I/O 花在一个数据单元上的时间是处理时间的 20 倍，这样的结果就不足为奇了。

表中所列并非真实数据，目的只在说明相对时间度量，现实情况绝非如此简单。正如您所看到的，影响应用程序执行效率的限定性因素，往往并非处理速率，而是 I/O。程序员热衷于调试代码，I/O 性能的调试往往被摆在第二位，甚至完全忽略。殊不知，在 I/O 性能上的小小投入就可换来可观的回报，想来实在令人惋惜。

1.2 CPU已不再是束缚

Java 程序员把全部精力用在优化处理效率上，而对 I/O 关注不足，在某种程度上讲这并非他们的错。在 Java 的早期，JVM 在解释字节码时往往很少或没有运行时优化。这就意味着，Java 程序往往拖得很长，其运行速率大大低于本地编译代码，因而对操作系统 I/O 子系统的要求并不太高。

如今在运行时优化方面，JVM 已然前进了一大步。现在 JVM 运行字节码的速率已经接近本地编译代码，借助动态运行时优化，其表现甚至还有所超越。这就意味着，多数 Java 应用程序已不再受 CPU 的束缚（把大量时间用在执行代码上），而更多时候是受 I/O 的束缚（等待数据传输）。

然而，在大多数情况下，Java 应用程序并非真的受着 I/O 的束缚。操作系统并非不能快速传送数据，让 Java 有事可做；相反，是 JVM 自身在 I/O 方面效率欠佳。操作系统与 Java 基于流的 I/O 模型有些不匹配。操作系统要移动的是大块数据（缓冲区），这往往是在硬件直接存储器存取（DMA）的协助下完成的。而 JVM 的 I/O 类喜欢操作小块数据——单个字节、几行文本。结果，操作系统送来整缓冲区的数据，`java.io` 的流数据类再花大量时间把它们拆成小块，往往拷贝一个小块就要往返于几层对象。操作系统喜欢整卡车地运来数据，`java.io` 类则喜欢一铲子一铲子地加工数据。有了 NIO，就可以轻松地把一卡车数据备份到您能直接使用的地方（*ByteBuffer* 对象）。

这并不是说使用传统的 I/O 模型无法移动大量数据——当然可以（现在依然可以）。具体地说，*RandomAccessFile* 类在这方面的效率就不低，只要坚持使用基于数组的 *read( )*和 *write( )*方法。这些方法与底层操作系统调用相当接近，尽管必须保留至少一份缓冲区拷贝。

如表 1-1 所示，如果您的代码大部分时间都处于 I/O 等待状态，那么，该考虑一下提升 I/O 效率的问题了，否则，您精心打造的代码多数时间都得闲着。

1.3 进入正题

操作系统研发人员将大量精力投入到提升 I/O 性能上。众多高手日以继夜地工作，只为完善数据传输技术。操作系统开发商为了取得竞争优势，投入大量时间、金钱，以便在测试数据上胜过竞争对手。

当今的操作系统是现代软件工程的奇迹（没错，有的比奇迹还奇迹），可是 Java 程序员如何能够既利用操作系统的强大功能，又保持平台独立性？唉，天下没有免费的午餐，此为一例。

JVM 是把双刃剑。它提供了统一的操作环境，让 Java 程序员不用再为操作系统环境的区别而烦恼。与特定平台相关的细枝末节大都被隐藏了起来，因而代码写得又快又容易。但是隐藏操作系统的技术细节也意味着某些个性鲜明、功能强大的特性被挡在了门外。

怎么办呢？如果您是程序员，可以使用 Java 本地接口（JNI）编写本地代码，直接使用操作系统特性。这样的话，您就被绑定在该操作系统上（也许还是其特定版本上）。如果您的本地代码不是 100%无漏洞，您还可能把 JVM 置于频繁出错乃至崩溃的境地。如果您是操作系统开发商，则可以在您的 JVM 实现中包含本地代码，以 Java API 的形式提供这些特性。但这样做可能违反您所签署相关许可协议，根据协议，您只能提供符合一致性要求的 JVM。Sun 曾就此问题将 Microsoft 告上法庭，因为很明显，JDirect 软件包只能在微软的系统上运行。如果以上方法都行不通，那么您只好转向其他语言，以实现对性能要求极为苛刻的应用。

为了解决这一问题，`java.nio` 软件包提供了新的抽象。具体地说，就是 *Channel* 和 *Selector* 类。它们提供了使用 I/O 服务的通用 API，JDK 1.4 以前的版本是无法使用这些服务的。天下还是没有免费的午餐：您无法使用每一种操作系统的每一种特性，但是这些新类还是提供了强大的新框架，涵盖了当今商业操作系统普遍提供的高效 I/O 特性。不仅如此，`java.nio.channels.spi` 还提供了新的服务提供接口（SPI），允许接入新型通道和选择器，同时又不违反规范的一致性。

随着 NIO 的面世，Java 已经为严肃的商业、娱乐、科研和学术应用做好了准备。在这些领域，高性能 I/O 是必不可少的。

除了 NIO，JDK 1.4 还包含许多其他重要改进。从 1.4 版开始，Java 平台已进入高度成熟期，它仍无法涉足的应用领域已所剩无几。David Flanagan 所著《Java 技术手册》（第四版）（*Java in a Nutshell, Fourth Edition* [O'Reilly]）是全面了解 JDK 1.4 各方面特性的绝佳向导。

1.4 I/O概念

Java 平台提供了一整套 I/O 隐喻，其抽象程度各有不同。然而，离冰冷的现实越远，要想搞清楚来龙去脉就越难——不管使用哪一种抽象，情况都是如此。JDK 1.4 的 NIO 软件包引入了一套新的抽象用于 I/O 处理。与以往不同的是，新的抽象把重点放在了如何缩短抽象与现实之间的距离上面。NIO 抽象与现实中存在的实体有着非常真实直接的交互关系。要想最大限度地满足 Java 应用程序的密集 I/O 需求，理解这些新的抽象，以及与其发生交互作用的 I/O 服务（其重要性并不亚于抽象），正是关键所在。

本书假定您熟知基本的 I/O 概念，因此，本节将快速回顾一些基本概念，为下一步论述新的 NIO 类如何运作奠定基础。NIO 类模拟 I/O 函数，因此，必须掌握操作系统层面的处理细节，才能理解新的 I/O 模型。

在阅读本书的过程中，理解以下概念是非常重要的：

- 缓冲区操作
- 内核空间与用户空间
- 虚拟内存
- 分页技术
- 面向文件的 I/O 和流 I/O
- 多工 I/O（就绪性选择）

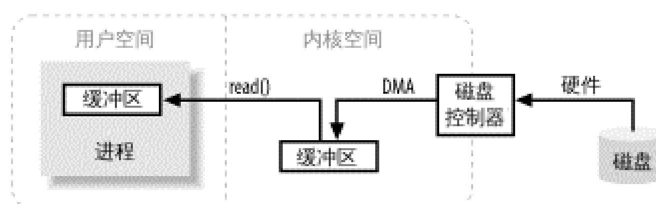
1.4.1 缓冲区操作

缓冲区，以及缓冲区如何工作，是所有 I/O 的基础。所谓“输入 / 输出”讲的无非就是把数据移进或移出缓冲区。

进程执行 I/O 操作，归结起来，也就是向操作系统发出请求，让它要么把缓冲区里的数据排干（写），要么用数据把缓冲区填满（读）。进程使用这一机制处理所有数据进出操作。操作系统内部处理这一任务的机制，其复杂程度可能超乎想像，但就概念而言，却非常直白易懂。

图 1-1 简单描述了数据从外部磁盘向运行中的进程的内存区域移动的过程。进程使用 *read( )*系统调用，要求其缓冲区被填满。内核随即向磁盘控制硬件发出命令，要求其从磁盘读取数据。磁盘控制器把数据直接写入内核内存缓冲区，这一步通过 DMA 完成，无需主 CPU 协助。一旦磁盘控制器把缓冲区装满，内核即把数据从内核空间的临时缓冲区拷贝到进程执行 *read( )*调用时指定的缓冲区。

图 1-1. I/O 缓冲区操作简图



图中明显忽略了很多细节，仅显示了涉及到的基本步骤。

注意图中用户空间和内核空间的概念。用户空间是常规进程所在区域。JVM 就是常规进程，驻守于用户空间。用户空间是非特权区域：比如，在该区域执行的代码就不能直接访问硬件设备。内核空间是操作系统所在区域。内核代码有特别的权力：它能与设备控制器通讯，控制着用户区域进程的运行状态，等等。最重要的是，所有 I/O 都直接（如这里所述）或间接（见 1.4.2 小节）通过内核空间。

当进程请求 I/O 操作的时候，它执行一个系统调用（有时称为陷阱）将控制权移交给内核。C/C++程序员所熟知的底层函数 *open( )*、*read( )*、*write( )*和 *close( )*要做的无非就是建立和执行适当的系统调用。当内核以这种方式被调用，它随即采取任何必要步骤，找到进程所需数据，并把数据传送到用户空间内的指定缓冲区。内核试图对数据进行高速缓存或预读取，因此进程所需数据可能已经在内核空间里了。如果是这样，该数据只需简单地拷贝出来即可。如果数据不在内核空间，则进程被挂起，内核着手把数据读进内存。

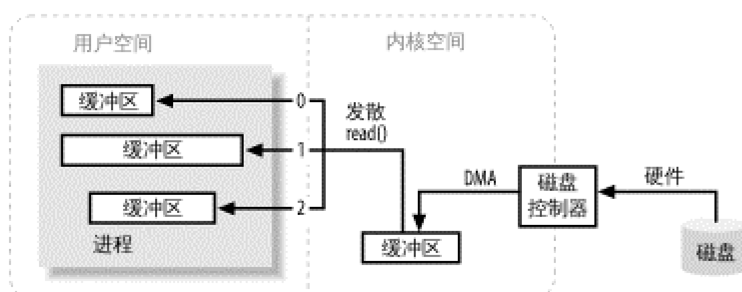
看了图 1-1，您可能会觉得，把数据从内核空间拷贝到用户空间似乎有些多余。为什么不直接让磁盘控制器把数据送到用户空间的缓冲区呢？这样做有几个问题。首先，硬件通常不能直接访问用户空间 [1]。其次，像磁盘这样基于块存储的硬件设备操作的是固定大小的数据块，而用户进程请求的可能是任意大小的或非对齐的数据块。在数据往来于用户空间与存储设备的过程中，内核负责数据的分解、再组合工作，因此充当着中间人的角色。

1.4.1.1 发散 / 汇聚

许多操作系统能把组装 / 分解过程进行得更加高效。根据发散 / 汇聚的概念，进程只需一个系统调用，就能把一连串缓冲区地址传递给操作系统。然后，内核就可以顺序填充或排干多个缓冲区，读的时候就把数据发散到多个用户空间缓冲区，写的时候再从多个缓冲区把数据汇聚起来（图 1-2）。

1 关于这一点，原因很多，不在本书讨论范围之内。硬件设备通常不能直接使用虚拟内存地址。

图 1-2. 三个缓冲区的发散读操作



这样用户进程就不必多次执行系统调用（那样做可能代价不菲），内核也可以优化数据的处理过程，因为它已掌握待传输数据的全部信息。如果系统配有多个 CPU，甚至可以同时填充或排干多个缓冲区。

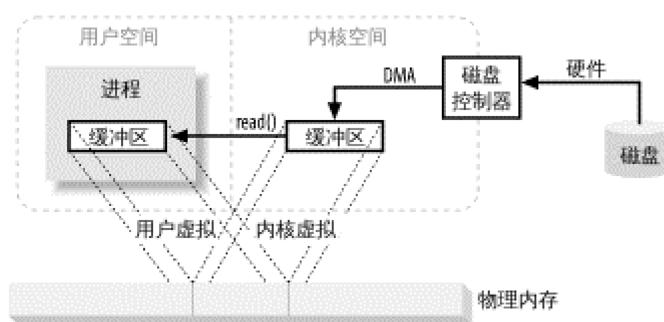
1.4.2 虚拟内存

所有现代操作系统都使用虚拟内存。虚拟内存意为使用虚假（或虚拟）地址取代物理（硬件 RAM）内存地址。这样做好处颇多，总结起来可分为两大类：

1. 一个以上的虚拟地址可指向同一个物理内存地址。
2. 虚拟内存空间可大于实际可用的硬件内存。

前一节提到，设备控制器不能通过 DMA 直接存储到用户空间，但通过利用上面提到的第一项，则可以达到相同效果。把内核空间地址与用户空间的虚拟地址映射到同一个物理地址，这样，DMA 硬件（只能访问物理内存地址）就可以填充对内核与用户空间进程同时可见的缓冲区（见图 1-3）。

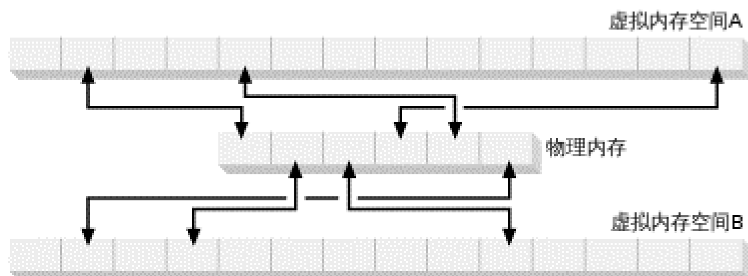
图 1-3. 内存空间多重映射



这样真是太好了，省去了内核与用户空间的往来拷贝，但前提条件是，内核与用户缓冲区必须使用相同的页对齐，缓冲区的大小还必须是磁盘控制器块大小（通常为 512 字节磁盘扇区）的倍数。操作系统把内存地址空间划分为页，即固定大小的字节组。内存页的大小总是磁盘块大小的倍数，通常为 2 次幂（这样可简化寻址操作）。典型的内存页为 1,024、2,048 和 4,096 字节。虚拟和物理内存页的大小总是相同的。图 1-4 显示了来自多个虚拟地址的虚拟内存页是如何映射到物理内

存的。

图 1-4. 内存页

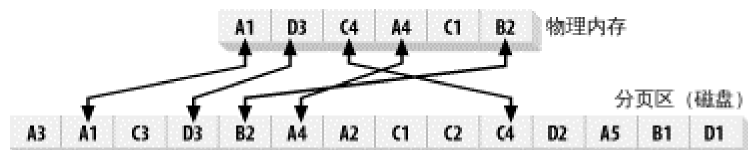


1.4.3 内存页面调度

为了支持虚拟内存的第二个特性（寻址空间大于物理内存），就必须进行虚拟内存分页（经常称为交换，虽然真正的交换是在进程层面完成，而非页层面）。依照该方案，虚拟内存空间的页面能够继续存在于外部磁盘存储，这样就为物理内存中的其他虚拟页面腾出了空间。从本质上说，物理内存充当了分页区的高速缓存；而所谓分页区，即从物理内存置换出来，转而存储于磁盘上的内存页面。

图 1-5 显示了分属于四个进程的虚拟页面，其中每个进程都有属于自己的虚拟内存空间。进程 A 有五个页面，其中两个装入内存，其余存储于磁盘。

图 1-5. 用于分页区高速缓存的物理内存



把内存页大小设定为磁盘块大小的倍数，这样内核就可直接向磁盘控制硬件发布命令，把内存页写入磁盘，在需要时再重新装入。结果是，所有磁盘 I/O 都在页层面完成。对于采用分页技术的现代操作系统而言，这也是数据在磁盘与物理内存之间往来的唯一方式。

现代 CPU 包含一个称为内存管理单元（MMU）的子系统，逻辑上位于 CPU 与物理内存之间。该设备包含虚拟地址向物理内存地址转换时所需映射信息。当 CPU 引用某内存地址时，MMU 负责确定该地址所在页（往往通过对地址值进行移位或屏蔽位操作实现），并将虚拟页号转换为物理页号（这一步由硬件完成，速度极快）。如果当前不存在与该虚拟页形成有效映射的物理内存页，MMU 会向 CPU 提交一个页错误。

页错误随即产生一个陷阱（类似于系统调用），把控制权移交给内核，附带导致错误的虚拟地址信息，然后内核采取步骤验证页的有效性。内核会安排页面调入操作，把缺失的页内容读回物理内存。这往往导致别的页被移出物理内存，好给新来的页让地方。在这种情况下，如果待移出的页

已经被碰过了（自创建或上次页面调入以来，内容已发生改变），还必须首先执行页面调出，把页内容拷贝到磁盘上的分页区。

如果所要求的地址不是有效的虚拟内存地址（不属于正在执行的进程的任何一个内存段），则该页不能通过验证，段错误随即产生。于是，控制权转交给内核的另一部分，通常导致的结果就是进程被强令关闭。

一旦出错的页通过了验证，MMU 随即更新，建立新的虚拟到物理的映射（如有必要，中断被移出页的映射），用户进程得以继续。造成页错误的用户进程对此不会有丝毫察觉，一切都在不知不觉中进行。

1.4.4 文件I/O

文件 I/O 属文件系统范畴，文件系统与磁盘迥然不同。磁盘把数据存在扇区上，通常一个扇区 512 字节。磁盘属硬件设备，对何谓文件一无所知，它只是提供了一系列数据存取窗口。在这点上，磁盘扇区与内存页颇有相似之处：都是统一大小，都可作为大的数组被访问。

文件系统是更高层次的抽象，是安排、解释磁盘（或其他随机存取块设备）数据的一种独特方式。您所写代码几乎无一例外地要与文件系统打交道，而不是直接与磁盘打交道。是文件系统定义了文件名、路径、文件、文件属性等抽象概念。

前一节讲到，所有 I/O 都是通过请求页面调度完成的。您应该还记得，页面调度是非常底层的操作，仅发生于磁盘扇区与内存页之间的直接传输。而文件 I/O 则可以任意大小、任意定位。那么，底层的页面调度是如何转换为文件 I/O 的？

文件系统把一连串大小一致的数据块组织到一起。有些块存储元信息，如空闲块、目录、索引等的映射，有些包含文件数据。单个文件的元信息描述了哪些块包含文件数据、数据在哪里结束、最后一次更新是什么时候，等等。

当用户进程请求读取文件数据时，文件系统需要确定数据具体在磁盘什么位置，然后着手把相关磁盘扇区读进内存。老式的操作系统往往直接向磁盘驱动器发布命令，要求其读取所需磁盘扇区。而采用分页技术的现代操作系统则利用请求页面调度取得所需数据。

操作系统还有个页的概念，其大小或者与基本内存页一致，或者是其倍数。典型的操作系统页从 2,048 到 8,192 字节不等，且始终是基本内存页大小的倍数。

采用分页技术的操作系统执行 I/O 的全过程可总结为以下几步：

- 确定请求的数据分布在文件系统的哪些页（磁盘扇区组）。磁盘上的文件内容和元数据可能跨越多个文件系统页，而且这些页可能也不连续。
- 在内核空间分配足够数量的内存页，以容纳得到确定的文件系统页。

- 在内存页与磁盘上的文件系统页之间建立映射。
- 为每一个内存页产生页错误。
- 虚拟内存系统俘获页错误，安排页面调入，从磁盘上读取页内容，使页有效。
- 一旦页面调入操作完成，文件系统即对原始数据进行解析，取得所需文件内容或属性信息。

需要注意的是，这些文件系统数据也会同其他内存页一样得到高速缓存。对于随后发生的 I/O 请求，文件数据的部分或全部可能仍旧位于物理内存当中，无需再从磁盘读取即可重复使用。

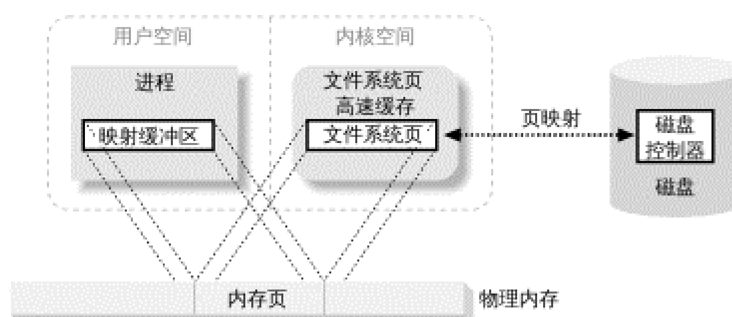
大多数操作系统假设进程会继续读取文件剩余部分，因而会预读额外的文件系统页。如果内存争用情况不严重，这些文件系统页可能在相当长的时间内继续有效。这样的话，当稍后该文件又被相同或不同的进程再次打开，可能根本无需访问磁盘。这种情况您可能也碰到过：当重复执行类似的操作，如在几个文件中进行字符串检索，第二遍运行得似乎快多了。

类似的步骤在写文件数据时也会采用。这时，文件内容的改变（通过 *write( )*）将导致文件系统页变脏，随后通过页面调出，与磁盘上的文件内容保持同步。文件的创建方式是，先把文件映射到空闲文件系统页，在随后的写操作中，再将文件系统页刷新到磁盘。

1.4.4.1 内存映射文件

传统的文件 I/O 是通过用户进程发布 *read( )*和 *write( )*系统调用来传输数据的。为了在内核空间的文件系统页与用户空间的内存区之间移动数据，一次以上的拷贝操作几乎总是免不了的。这是因为，在文件系统页与用户缓冲区之间往往没有一一对应关系。但是，还有一种大多数操作系统都支持的特殊类型的 I/O 操作，允许用户进程最大限度地利用面向页的系统 I/O 特性，并完全摒弃缓冲区拷贝。这就是内存映射 I/O，如图 1-6 所示。

图 1-6. 用户内存到文件系统页的映射



内存映射 I/O 使用文件系统建立从用户空间直到可用文件系统页的虚拟内存映射。这样做有几个好处：

- 用户进程把文件数据当作内存，所以无需发布 *read( )*或 *write( )*系统调用。
- 当用户进程碰触到映射内存空间，页错误会自动产生，从而将文件数据从磁盘读进内

存。如果用户修改了映射内存空间，相关页会自动标记为脏，随后刷新到磁盘，文件得到更新。

- 操作系统的虚拟内存子系统会对页进行智能高速缓存，自动根据系统负载进行内存管理。
- 数据总是按页对齐的，无需执行缓冲区拷贝。
- 大型文件使用映射，无需耗费大量内存，即可进行数据拷贝。

虚拟内存和磁盘 I/O 是紧密关联的，从很多方面看来，它们只是同一件事物的两面。在处理大量数据时，尤其要记得这一点。如果数据缓冲区是按页对齐的，且大小是内建页大小的倍数，那么，对大多数操作系统而言，其处理效率会大幅提升。

1.4.4.2 文件锁定

文件锁定机制允许一个进程阻止其他进程存取某文件，或限制其存取方式。通常的用途是控制共享信息的更新方式，或用于事务隔离。在控制多个实体并行访问共同资源方面，文件锁定是必不可少的。数据库等复杂应用严重信赖于文件锁定。

“文件锁定”从字面上看有锁定整个文件的意思（通常的确是那样），但锁定往往可以发生在更为细微的层面，锁定区域往往可以细致到单个字节。锁定与特定文件相关，开始于文件的某个特定字节地址，包含特定数量的连续字节。这对于协调多个进程互不影响地访问文件不同区域，是至关重要的。

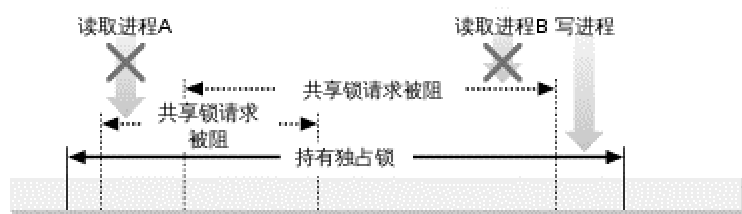
文件锁定有两种方式：共享的和独占的。多个共享锁可同时对同一文件区域发生作用；独占锁则不同，它要求相关区域不能有其他锁定在起作用。

共享锁和独占锁的经典应用，是控制最初用于读取的共享文件的更新。某个进程要读取文件，会先取得该文件或该文件部分区域的共享锁。第二个希望读取相同文件区域的进程也会请求共享锁。两个进程可以并行读取，互不影响。但是，假如有第三个进程要更新该文件，它会请求独占锁。该进程会处于阻滞状态，直到既有锁定（共享的、独占的）全部解除。一旦给予独占锁，其他共享锁的读取进程会处于阻滞状态，直到独占锁解除。这样，更新进程可以更改文件，而其他读取进程不会因为文件的更改得到前后不一致的结果。图 1-7 和图 1-8 描述了这一过程。

图 1-7. 共享锁阻断独占锁请求



图 1-8. 独占锁阻断共享锁请求



文件锁有建议使用和强制使用之分。建议型文件锁会向提出请求的进程提供当前锁定信息，但操作系统并不要求一定这样做，而是由相关进程进行协调并关注锁定信息。多数 Unix 和类 Unix 操作系统使用建议型锁，有些也使用强制型锁或兼而有之。

强制型锁由操作系统或文件系统强行实施，不管进程对锁的存在知道与否，都会阻止其对文件锁定区域的访问。微软的操作系统往往使用的是强制型锁。假定所有文件锁均为建议型，并在访问共同资源的各个应用程序间使用一致的文件锁定，是明智之举，也是唯一可行的跨平台策略。依赖于强制文件锁定的应用程序，从根子上讲就是不可移植的。

1.4.5 流I/O

并非所有 I/O 都像前几节讲的是面向块的，也有流 I/O，其原理模仿了通道。I/O 字节流必须顺序存取，常见的例子有 TTY（控制台）设备、打印机端口和网络连接。

流的传输一般（也不必然如此）比块设备慢，经常用于间歇性输入。多数操作系统允许把流置于非块模式，这样，进程可以查看流上是否有输入，即便当时没有也不影响它干别的。这样一种能力使得进程可以在有输入的时候进行处理，输入流闲置的时候执行其他功能。

比非块模式再进一步，就是就绪性选择。就绪性选择与非块模式类似（常常就是建立在非块模式之上），但是把查看流是否就绪的任务交给了操作系统。操作系统受命查看一系列流，并提醒进程哪些流已经就绪。这样，仅仅凭借操作系统返回的就绪信息，进程就可以使用相同代码和单一线程，实现多活动流的多路传输。这一技术广泛用于网络服务器领域，用来处理数量庞大的网络连接。就绪性选择在大容量缩放方面是必不可少的。

1.5 总结

本章概述了系统层面的 I/O，只是一带而过，肯定很不全面。如果需要更加详尽地了解相关内容，可以找本好的参考书，有不少的。我过去的老板 Avi Silberschatz 所著《操作系统概念》（第六版）（*Operating System Concepts, Sixth Edition* [John Wiley & Sons]）是本权威的操作系统教科书，从这本书出发是个不错的选择。

通过前面的讲述，您应当对后面各章所涵盖内容有了大致的了解。带上您的知识储备，随我前往核心主题：Java New I/O（NIO）。学习了新的 NIO 抽象，也别忘了脑子里的具体概念。理解了这些基本概念，才能更容易地识别新类所模拟之 I/O 性能。

NIO 学习之旅即将启程。发动机在轰鸣，一切都已就绪。快上来吧，找个位子，舒舒服服坐好，让我们立刻开拔。

第二章 缓冲区

一切都是相对的。

——伟大的阿尔伯特·爱因斯坦

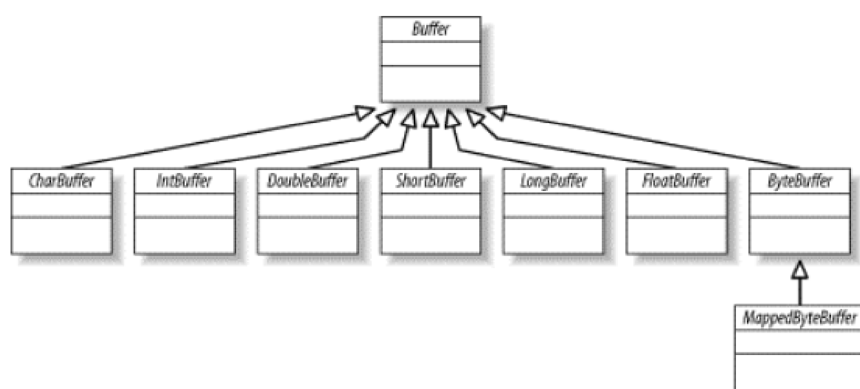
我们以 Buffer 类开始我们对 java.nio 软件包的浏览历程。这些类是 java.nio 的构造基础。在本章中，我们将深入研究缓冲区，了解各种不同的类型，并学会怎样使用。到那时我们将明了 java.nio 缓冲区是如何与 java.nio.channels 这一通道类相联系的。

一个Buffer对象是固定数量的数据的容器。其作用是一个存储器，或者分段运输区，在这里数据可被存储并在之后用于检索。缓冲区如我们在第一章所讨论的那样被写满和释放。对于每个非布尔原始数据类型都有一个缓冲区类。尽管缓冲区作用于它们存储的原始数据类型，但缓冲区十分倾向于处理字节。非字节缓冲区可以在后台执行从字节或到字节的转换，这取决于缓冲区是如何创建的[2]。我们将在本章节后面的部分检查缓冲区数据存储的含义。

缓冲区的工作与通道紧密联系。通道是 I/O 传输发生时通过的入口，而缓冲区是这些数据传输的来源或目标。对于离开缓冲区的传输，您想传递出去的数据被置于一个缓冲区，被传送到通道。对于传回缓冲区的传输，一个通道将数据放置在您所提供的缓冲区中。这种在协同对象（通常是您所写的对象以及一到多个 Channel 对象）之间进行的缓冲区数据传递是高效数据处理的关键。通道将在第三章被详细涉及。

图 2-1 是 Buffer 的类层次图。在顶部是通用 Buffer 类。Buffer 定义所有缓冲区类型共有的操作，无论是它们所包含的数据类型还是可能具有的特定行为。这一共同点将会成为我们的出发点。

图 2-1. Buffer 类的家谱



[2] 这暗含了按字节排序的问题，我们将在 2.4.1 部分对此进行讨论。

2.1 缓冲区基础

概念上，缓冲区是包在一个对象内的基本数据元素数组。Buffer 类相比一个简单数组的优点是它将关于数据的数据内容和信息包含在一个单一的对象中。Buffer 类以及它专有的子类定义了一个用于处理数据缓冲区的 API。

2.1.1 属性

所有的缓冲区都具有四个属性来提供关于其所包含的数据元素的信息。它们是：

容量（Capacity）

缓冲区能够容纳的数据元素的最大数量。这一容量在缓冲区创建时被设定，并且永远不能被改变。

上界（Limit）

缓冲区的第一个不能被读或写的元素。或者说，缓冲区中现存元素的计数。

位置（Position）

下一个要被读或写的元素的索引。位置会自动由相应的 *get( )*和 *put( )*函数更新。

标记（Mark）

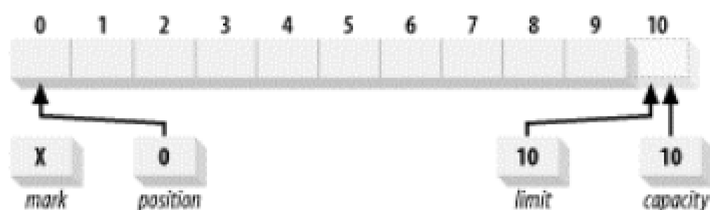
一个备忘位置。调用 *mark( )*来设定 mark = postion。调用 *reset( )*设定 position = mark。标记在设定前是未定义的(undefined)。

这四个属性之间总是遵循以下关系：

0 <= mark <= position <= limit <= capacity

让我们来看看这些属性在实际应用中的一些例子。图 2-2 展示了一个新创建的容量为 10 的 ByteBuffer 逻辑视图。

图 2-2. 新创建的 ByteBuffer



位置被设为 0，而且容量和上界被设为 10，刚好经过缓冲区能够容纳的最后一个字节。标记最初未定义。容量是固定的，但另外的三个属性可以在使用缓冲区时改变。

2.1.2 缓冲区 API

让我们来看一下可以如何使用一个缓冲区。以下是 Buffer 类的方法签名：

```
package java.nio;

public abstract class Buffer {
      public final int capacity( )
      public final int position( )
      public final Buffer position (int newPositio
      public final int limit( )
      public final Buffer limit (int newLimit)
      public final Buffer mark( )
      public final Buffer reset( )
      public final Buffer clear( )
      public final Buffer flip( )
      public final Buffer rewind( )
      public final int remaining( )
      public final boolean hasRemaining( )
      public abstract boolean isReadOnly( );
}
```

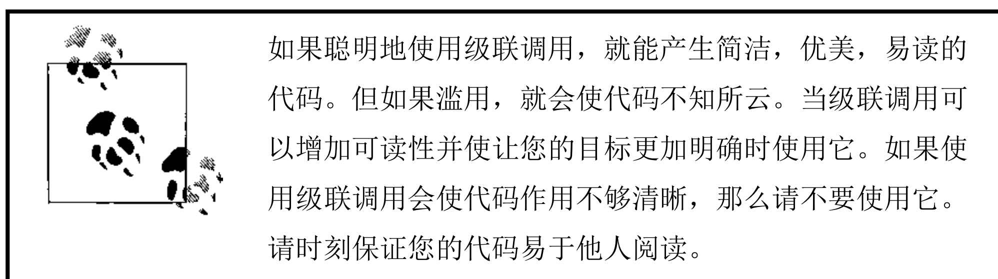
关于这个 API 有一点要注意的是，像 clear()这类函数，您通常应当返回 void，而不是 Buffer 引用。这些函数将引用返回到它们在（this）上被引用的对象。这是一个允许级联调用的类设计方法。级联调用允许这种类型的代码：

```
buffer.mark( );
buffer.position(5);
buffer.reset( );
```

被简写为：

```
buffer.mark().position(5).reset( );
```

java.nio 中的类被特意地设计为支持级联调用。您可能已经在 StringBuffer 类中看到了级联调用的使用。



如果聪明地使用级联调用，就能产生简洁，优美，易读的代码。但如果滥用，就会使代码不知所云。当级联调用可以增加可读性并使让您的目标更加明确时使用它。如果使用级联调用会使代码作用不够清晰，那么请不要使用它。请时刻保证您的代码易于他人阅读。

对于 API 还要注意的一点是 isReadOnly()函数。所有的缓冲区都是可读的，但并非所有都可写。每个具体的缓冲区类都通过执行 isReadOnly()来标示其是否允许该缓存区的内容被修改。一些类型的缓冲区类可能未使其数据元素存储在一个数组中。例如 MappedByteBuffer 的内容可能实际是一个只读文件。您也可以明确地创建一个只读视图缓冲区，来防止对内容的意外修改。对只读的缓冲区的修改尝试将会导致 ReadOnlyBufferException 抛出。但是我们要提前做好准备。

2.1.3 存取

让我们从起点开始。缓冲区管理着固定数目的数据元素。但在任何特定的时刻，我们可能只对缓冲区中的一部分元素感兴趣。换句话说，在我们想清空缓冲区之前，我们可能只使用了缓冲区的一部分。这时，我们需要能够追踪添加到缓冲区内的数据元素的数量，放入下一个元素的位置等等的方法。位置属性做到了这一点。它在调用 put()时指出了下一个数据元素应该被插入的位置，或者当 get()被调用时指出下一个元素应从何处检索。聪明的读者会注意到上文所列出的的 Buffer API 并没有包括 get()或 put()函数。每一个 Buffer 类都有这两个函数，但它们所采用的参数类型，以及它们返回的数据类型，对每个子类来说都是唯一的，所以它们不能在顶层 Buffer 类中被抽象地声明。它们的定义必须被特定类型的子类所遵从。对于这一讨论，我们将假设使用具有这里所给出的函数的 ByteBuffer 类（get()和 put()还有更多的形式，我们将在 2.1.10 小节中进行讨论）：

```
public abstract class ByteBuffer
      extends Buffer implements Comparable
{
      // This is a partial API listing

      public abstract byte get( );
      public abstract byte get (int index);
      public abstract ByteBuffer put (byte b);
      public abstract ByteBuffer put (int index, byte b);
}
```

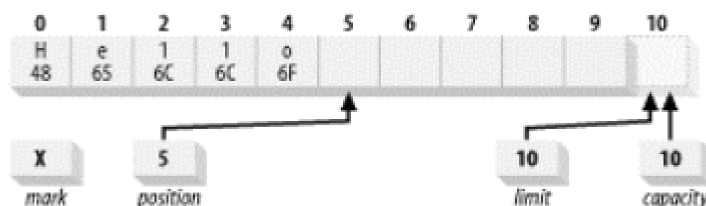
Get 和 put 可以是相对的或者是绝对的。在前面的程序列表中，相对方案是不带有索引参数的函数。当相对函数被调用时，位置在返回时前进一。如果位置前进过多，相对运算就会抛出异常。对于 put()，如果运算会导致位置超出上界，就会抛出 BufferOverflowException 异常。对于 get()，如果位置不小于上界，就会抛出 BufferUnderflowException 异常。绝对存取不会影响缓冲区的位置属性，但是如果您所提供的索引超出范围（负数或不小于上界），也将抛出 IndexOutOfBoundsException 异常。

2.1.4 填充

让我们看一个例子。我们将代表“Hello”字符串的 ASCII 码载入一个名为 buffer 的 ByteBuffer 对象中。当在图 2.2 所新建的缓冲区上执行以下代码后，缓冲区的结果状态如图 2.3 所示：

```
buffer.put((byte)'H').put((byte)'e').put((byte)'l').put((byte)'l').put((byte)'o');
```

图 2-3 五次调用 put()之后的缓冲区



注意本例中的每个字符都必须被强制转换为 byte。我们不能不经强制转换而这样操做：

```
buffer.put('H');
```

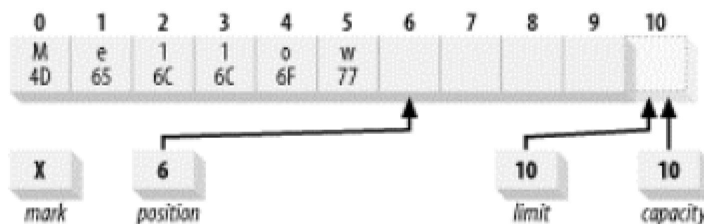
因为我们存放的是字节而不是字符。记住在 java 中，字符在内部以 Unicode 码表示，每个 Unicode 字符占 16 位。本章节的例子使用包含 ascii 字符集数值的字节。通过将 char 强制转换为 byte，我们删除了前八位来建立一个八位字节值。这通常只适合于拉丁字符而不能适合所有可能的 Unicode 字符。为了让事情简化，我们暂时故意忽略字符集的映射问题。第六章中将详细涉及字符编码。

既然我们已经在 buffer 中存放了一些数据，如果我们想在不丢失位置的情况下进行一些更改该怎么办呢？put()的绝对方案可以达到这样的目的。假设我们想将缓冲区中的内容从“Hello”的 ASCII 码更改为“Mellow”。我们可以这样实现：

```
buffer.put(0,(byte)'M').put((byte)'w');
```

这里通过进行一次绝对方案的 put 将 0 位置的字节代替为十六进制数值 0x4d，将 0x77 放入当前位置（当前位置不会受到绝对 put()的影响）的字节，并将位置属性加一。结果如图 2.4 所示。

图 2-4 修改后的 buffer



2.1.5 翻转

我们已经写满了缓冲区，现在我们必须准备将其清空。我们想把这个缓冲区传递给一个通道，以使内容能被全部写出。但如果通道现在在缓冲区上执行 get()，那么它将从我们刚刚插入的有用数据之外取出未定义数据。如果我们将位置值重新设为 0，通道就会从正确位置开始获取，但是它是怎样知道何时到达我们所插入数据末端的呢？这就是上界属性被引入的目的。上界属性指明了缓冲区有效内容的末端。我们需要将上界属性设置为当前位置，然后将位置重置为 0。我们可以人工用下面的代码实现：

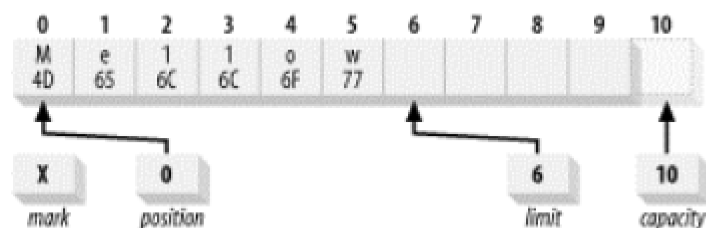
```
buffer.limit(buffer.position()).position(0);
```

但这种从填充到释放状态的缓冲区翻转是 API 设计者预先设计好的，他们为我们提供了一个非常便利的函数：

```
Buffer.flip();
```

Flip()函数将一个能够继续添加数据元素的填充状态的缓冲区翻转成一个准备读出元素的释放状态。在翻转之后，图 2.4 的缓冲区会变成图 2.5 中的样子。

图 2-5. 被翻转后的缓冲区



Rewind()函数与 flip()相似，但不影响上界属性。它只是将位置值设回 0。您可以使用 rewind()后退，重读已经被翻转的缓冲区中的数据。

如果将缓冲区翻转两次会怎样呢？它实际上会大小变为 0。按照图 2.5 的相同步骤对缓冲区进行操作；把上界设为位置的值，并把位置设为 0。上界和位置都变成 0。尝试对缓冲区上位置和上界都为 0 的 get()操作会导致 BufferUnderflowException 异常。而 put()则会导致 BufferOverflowException 异常。

2.1.6 释放

如果我们现在将图 2.5 中的缓冲区传入通道，它将取出我们存放在那里的数据，从位置开始直到上界结束。很简单，不是吗？

同样地，如果您接收到一个在别处被填满的缓冲区，您可能需要在检索内容之前将其翻转。例如，如果一个通道的 read()操作完成，而您想要查看被通道放入缓冲区内的数据，那么您需要在调用 get()之前翻转缓冲区。通道对象在缓冲区上调用 put()增加数据；put 和 read 可以随意混合使用。

布尔函数 hasRemaining()会在释放缓冲区时告诉您是否已经达到缓冲区的上界。以下是一种将数据元素从缓冲区释放到一个数组的方法(在 2.1.10 小节中，我们将学到进行批量传输的更高效的方法)。

```
for (int i = 0; buffer.hasRemaining(  ), i++) {
       myByteArray [i] = buffer.get(  );
}
```

作为选择，remaining()函数将告知您从当前位置到上界还剩余的元素数目。您也可以通过下面的循环来释放图 2-5 所示的缓冲区。

```
int count = buffer.remaining(  );

for (int i = 0; i < count, i++) {
       myByteArray [i] = buffer.get(  );
}
```

如果您对缓冲区有专门的控制，这种方法会更高效，因为上界不会在每次循环重复时都被检查（这要求调用一个 buffer 样例程序）。上文中的第一个例子允许多线程同时从缓冲区释放元素。



缓冲区并不是多线程安全的。如果您想以多线程同时存取特定的缓冲区，您需要在存取缓冲区之前进行同步（例如对缓冲区对象进行跟踪）。

一旦缓冲区对象完成填充并释放，它就可以被重新使用了。Clear()函数将缓冲区重置为空状态。它并不改变缓冲区中的任何数据元素，而是仅仅将上界设为容量的值，并把位置设回 0，如图 2.2 所示。这使得缓冲区可以被重新填入。参见示例 2.1。

例 2.1 填充和释放缓冲区

```
package com.ronsoft.books.nio.buffers;

import java.nio.CharBuffer;

/**
 * Buffer fill/drain example.  This code uses the simplest
 * means of filling and draining a buffer: one element at
 * a time.
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class BufferFillDrain
{
   public static void main (String [] argv)
      throws Exception
   {
      CharBuffer buffer = CharBuffer.allocate (100);

      while (fillBuffer (buffer)) {
         buffer.flip(  );
         drainBuffer (buffer);
         buffer.clear(  );
      }
   }

   private static void drainBuffer (CharBuffer buffer)
{
while (buffer.hasRemaining(  )) {
         System.out.print (buffer.get(  ));
      }

      System.out.println ("");
   }

   private static boolean fillBuffer (CharBuffer buffer)
   {
      if (index >= strings.length) {
         return (false);
      }

      String string = strings [index++];

      for (int i = 0; i < string.length(  ); i++) {
         buffer.put (string.charAt (i));
      }
```

```
        return (true);
    }

    private static int index = 0;

    private static String [] strings = {
        "A random string value",
        "The product of an infinite number of monkeys",
        "Hey hey we're the Monkees",
        "Opening act for the Monkees: Jimi Hendrix",
        "'Scuse me while I kiss this fly",  // Sorry Jimi ;-)
        "Help Me!  Help Me!",
    };
}
```

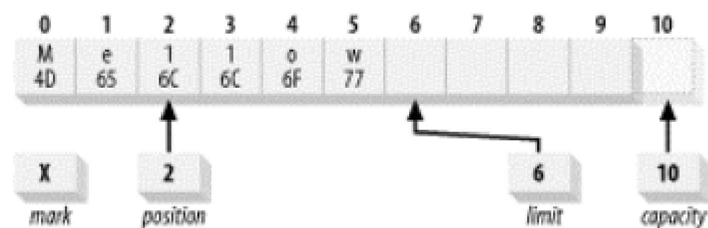
2.1.7 压缩

```
public abstract class ByteBuffer
        extends Buffer implements Comparable
{
        // This is a partial API listing

        public abstract ByteBuffer compact(  );
}
```

有时，您可能只想从缓冲区中释放一部分数据，而不是全部，然后重新填充。为了实现这一点，未读的数据元素需要下移以使第一个元素索引为 0。尽管重复这样做会效率低下，但这有时非常必要，而 API 对此为您提供了一个 compact()函数。这一缓冲区工具在复制数据时要比您使用 get()和 put()函数高效得多。所以当您需要时，请使用 compact()。图 2.6 显示了一个我们已经释放了一些元素，并且现在我们想要对其进行压缩的缓冲区。

图 2.6 被部分释放的缓冲区

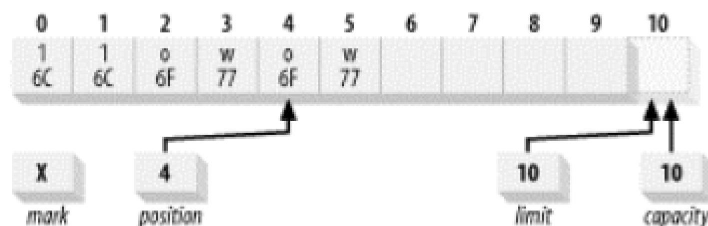


这样操作：

```
buffer.compact();
```

会导致缓冲区的状态如图 2-7 所示：

图 2.7 压缩后的 buffer



这里发生了几件事。您会看到数据元素 2-5 被复制到 0-3 位置。位置 4 和 5 不受影响，但现在正在或已经超出了当前位置，因此是“死的”。它们可以被之后的 put()调用重写。还要注意的是，位置已经被设为被复制的数据元素的数目。也就是说，缓冲区现在被定位在缓冲区中最后一个“存活”元素后插入数据的位置。最后，上界属性被设置为容量的值，因此缓冲区可以被再次填满。调用 compact()的作用是丢弃已经释放的数据，保留未释放的数据，并使缓冲区对重新填充容量准备就绪。

您可以用这种类似于先入先出（FIFO）队列的方式使用缓冲区。当然也存在更高效的算法（缓冲区移位并不是一个处理队列的非常高效的方法）。但是压缩对于使缓冲区与您从端口中读入的数据（包）逻辑块流的同步来说也许是一种便利的方法。

如果您想在压缩后释放数据，缓冲区会像之前所讨论的那样需要被翻转。无论您之后是否要向缓冲区中添加新的数据，这一点都是必要的。

2.1.8 标记

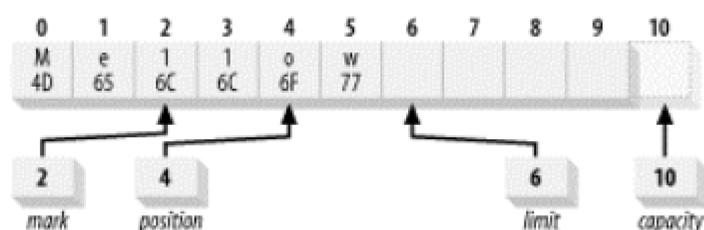
这本章节的开头，我们已经涉及了缓冲区四种属性中的三种。第四种，标记，使缓冲区能够记住一个位置并在之后将其返回。缓冲区的标记在 *mark( )*函数被调用之前是未定义的，调用时标记被设为当前位置的值。*reset( )*函数将位置设为当前的标记值。如果标记值未定义，调用 *reset( )*将导致 InvalidMarkException 异常。一些缓冲区函数会抛弃已经设定的标记（*rewind( )*，*clear( )*，以及 *flip( )*总是抛弃标记）。如果新设定的值比当前的标记小，调用 *limit( )*或 *position( )*带有索引参数的版本会抛弃标记。

注意不要混淆 *reset( )*和 *clear( )*。*clear( )*函数将清空缓冲区，而 *reset( )*位置返回到一个先前设定的标记。

让我们看看这是如何进行的。在图 2.5 的缓冲区上执行以下代码将会导致图 2-8 所显示的缓冲区状态。

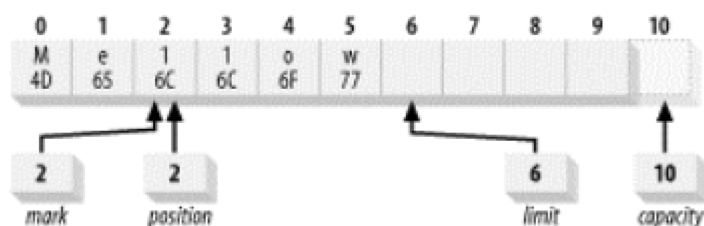
```
buffer.position(2).mark().position(4);
```

图 2-8 设有一个标记的缓冲区



如果这个缓冲区现在被传递给一个通道，两个字节（“ow”）将会被发送，而位置会前进到 6。如果我们此时调用 *reset( )*，位置将会被设为标记，如图 2-9 所示。再次将缓冲区传递给通道将导致四个字节（“llow”）被发送。

图 2-9 一个缓冲区位置被重设为标记



结果可能没什么意义（owllow 会被写入通道），但您了解了概念。

2.1.9 比较

有时候比较两个缓冲区所包含的数据是很有必要的。所有的缓冲区都提供了一个常规的 *equals( )*函数用以测试两个缓冲区的是否相等，以及一个 *compareTo( )*函数用以比较缓冲区。

```
public abstract class ByteBuffer
       extends Buffer implements Comparable
{
       // This is a partial API listing

       public boolean equals (Object ob)
       public int compareTo (Object ob)
}
```

两个缓冲区可用下面的代码来测试是否相等：

```
if (buffer1.equals (buffer2)) {
       doSomething( );
}
```

如果每个缓冲区中剩余的内容相同，那么 *equals( )*函数将返回 true，否则返回 false。因为这个测试是用于严格的相等而且是可换向的。前面的程序清单中的缓冲区名称可以颠倒，并会产生相同的结果。

两个缓冲区被认为相等的充要条件是：

- 两个对象类型相同。包含不同数据类型的 buffer 永远不会相等，而且 buffer 绝不会等于非 buffer 对象。
- 两个对象都剩余同样数量的元素。Buffer 的容量不需要相同，而且缓冲区中剩余数据的索引也不必相同。但每个缓冲区中剩余元素的数目（从位置到上界）必须相同。
- 在每个缓冲区中应被 Get()函数返回的剩余数据元素序列必须一致。

如果不满足以上任意条件，就会返回 false。

图 2-10 说明了两个属性不同的缓冲区也可以相等。

图 2-11 显示了两个相似的缓冲区，可能看起来是完全相同的缓冲区，但测试时会发现并不相等。

图 2-10 两个被认为是相等的缓冲区

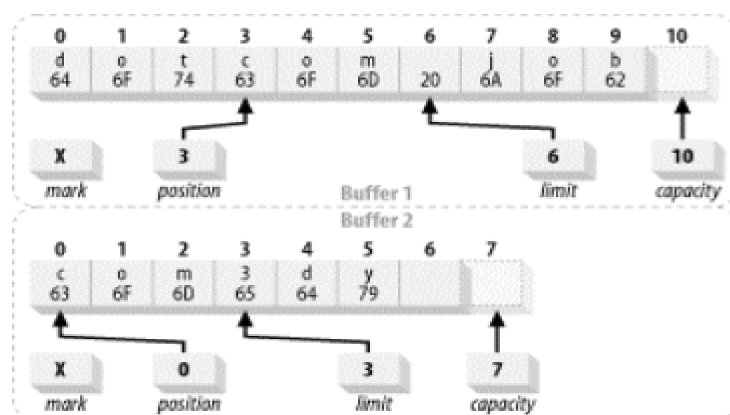
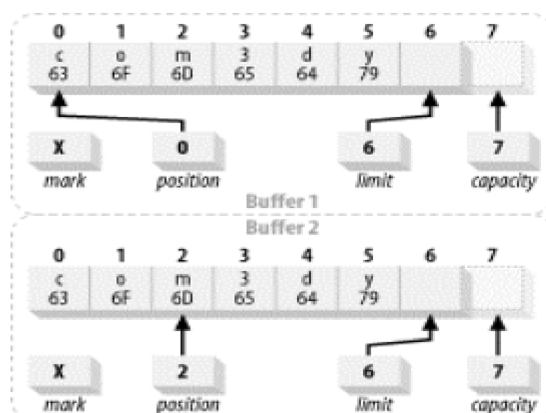


图 2-11 两个被认为不相等的缓冲区



缓冲区也支持用 *compareTo( )*函数以词典顺序进行比较。这一函数在缓冲区参数小于，等于，或者大于引用 *compareTo( )*的对象实例时，分别返回一个负整数，0 和正整数。这些就是所有典型的缓冲区所实现的 java.lang.Comparable 接口语义。这意味着缓冲区数组可以通过调用 java.util.Arrays.sort()函数按照它们的内容进行排序。

与 *equals( )*相似，*compareTo( )*不允许不同对象间进行比较。但 *compareTo( )*更为严格：如果您传递一个类型错误的对象，它会抛出 ClassCastException 异常，但 *equals( )*只会返回 false。

比较是针对每个缓冲区内剩余数据进行的，与它们在 *equals( )*中的方式相同，直到不相等的元素被发现或者到达缓冲区的上界。如果一个缓冲区在不相等元素发现前已经被耗尽，较短的缓冲区被认为是小于较长的缓冲区。不像 *equals( )*，*compareTo( )*不可交换：顺序问题。在本例中，一个小于零的结果表明 buffer2 小于 buffer1，而表达式的值就会是 true：

```
if (buffer1.compareTo (buffer2) < 0) {
       doSomething( );
}
```

如果前面的代码被应用到图 2-10 所示的缓冲区中，结果会是 0，而 if 语句将毫无用处。被应用到图 2-11 的缓冲区的相同测试将会返回一个正数（表明 buffer2 大于 buffer1），而这个表达式也会被判断为 false。

2.1.10 批量移动

缓冲区的涉及目的就是为了能够高效传输数据。一次移动一个数据元素，如例 2-1 所示的那样并不高效。如您在下面的程序清单中所看到的那样，buffer API 提供了向缓冲区内外批量移动数据元素的函数。

```
public abstract class CharBuffer
       extends Buffer implements CharSequence, Comparable
{
       // This is a partial API listing

       public CharBuffer get (char [] dst)
       public CharBuffer get (char [] dst, int offset, int length)

       public final CharBuffer put (char[] src)
       public CharBuffer put (char [] src, int offset, int length)
       public CharBuffer put (CharBuffer src)

       public final CharBuffer put (String src)
       public CharBuffer put (String src, int start, int end)
}
```

有两种形式的 *get( )*可供从缓冲区到数组进行的数据复制使用。第一种形式只将一个数组作为参数，将一个缓冲区释放到给定的数组。第二种形式使用 offset 和 length 参数来指定目标数组的子区间。这些批量移动的合成效果与前文所讨论的循环是相同的，但是这些方法可能高效得多，因为这种缓冲区实现能够利用本地代码或其他的优化来移动数据。

批量移动总是具有指定的长度。也就是说，您总是要求移动固定数量的数据元素。当参看程序签名时这一点还不明显，但是对 *get( )*的这一引用：

```
buffer.get(myArray);
```

等价于：

```
buffer.get(myArray,0,myArray.length);
```



批量传输的大小总是固定的。省略长度意味着*整个*数组会被填满。

如果您所要求的数量的数据不能被传送，那么不会有数据被传递，缓冲区的状态保持不变，同时抛出 BufferUnderflowException 异常。因此当您传入一个数组并且没有指定长度，您就相当于要求整个数组被填充。如果缓冲区中的数据不够完全填满数组，您会得到一个异常。这意味着如果您想将一个小型缓冲区传入一个大型数组，您需要明确地指定缓冲区中剩余的数据长度。上面的第一个例子不会如您第一眼所推出的结论那样，将缓冲区内剩余的数据元素复制到数组的底部。要将一个缓冲区释放到一个大数组中，要这样做：

```
char [] bigArray = new char [1000];

// Get count of chars remaining in the buffer
int length = buffer.remaining( );

// Buffer is known to contain < 1,000 chars
buffer.get (bigArrray, 0, length);

// Do something useful with the data
processData (bigArray, length);
```

记住在调用 *get( )*之前必须查询缓冲区中的元素数量（因为我们需要告知 *processData( )*被放置在 bigArray 中的字符个数）。调用 *get( )*会向前移动缓冲区的位置属性，所以之后调用 *remaining( )*会返回 0。*get( )*的批量版本返回缓冲区的引用，而不是被传送的数据元素的计数，以减轻级联调用的困难。

另一方面，如果缓冲区存有比数组能容纳的数量更多的数据，您可以重复利用如下文所示的程序块进行读取：

```
char [] smallArray = new char [10];

while (buffer.hasRemaining( )) {
       int length = Math.min (buffer.remaining( ), smallArray.length);

       buffer.get (smallArray, 0, length);
       processData (smallArray, length);
}
```

Put()的批量版本工作方式相似，但以相反的方向移动数据，从数组移动到缓冲区。他们在传送数据的大小方面有着相同的语义：

```
buffer.put(myArray);
```

等价于：

```
buffer.put(myArray,0,myArray.length);
```

如果缓冲区有足够的空间接受数组中的数据（buffer.remaining()>myArray.length），数据将会被复制到从当前位置开始的缓冲区，并且缓冲区位置会被提前所增加数据元素的数量。如果缓冲区中没有足够的空间，那么不会有数据被传递，同时抛出一个 BufferOverflowException 异常。

也可以通过调用带有一个缓冲区引用作为参数的 put()来在两个缓冲区内进行批量传递。

```
buffer.put(srcBuffer);
```

这等价于（假设 dstBuffer 有足够的空间）：

```
while (srcBuffer.hasRemaining( )) {
      dstBuffer.put (srcBuffer.get( ));
}
```

两个缓冲区的位置都会前进所传递的数据元素的数量。范围检查会像对数组一样进行。具体来说，如果 srcBuffer.remaining( )大于 dstBuffer.remaining( )，那么数据不会被传递，同时抛出 BufferOverflowException 异常。如果您对将一个缓冲区传递给它自己，就会引发 java.lang.IllegalArgumentException 异常。

在这一章节中我一直使用 CharBuffer 为例，而且到目前为止，这一讨论也已经应用到了其他的典型缓冲区上，比如 FloatBuffer，LongBuffer，等等。但是在下面的 API 程序清单的最后两个函数中包含了两个只对 CharBuffer 适用的批量移动函数。

```
public abstract class CharBuffer
      extends Buffer implements CharSequence, Comparable
{
      // This is a partial API listing

      public final CharBuffer put (String src)
      public CharBuffer put (String src, int start, int end)
}
```

这些函数使用 String 作为参数，而且与作用于 char 数组的批量移动函数相似。如所有的 java 程序员所知，String 不同于 char 数组。但 String 确实包含 char 字符串，而且我们人类确实倾向于将其在概念上认为是 char 数组（尤其是我们中曾经是或者现在还是 C 或 C++程序员的那些人）。由于这些原因，CharBuffer 类提供了将 String 复制到 CharBuffer 中的便利方法。

String 移动与 char 数组移动相似，除了在序列上是由 start 和 end+1 下标确定（与 String.subString()类似），而不是 start 下标和 length。所以：

```
buffer.put(myString);
```

等价于：

```
buffer.put(myString,0,myString.length);
```

而这就是您怎样复制字符 5-8，总共四个字符，从 myString 复制到 buffer。

```
buffer.put(myString,5,9);
```

String 批量移动等效于下面的代码：

```
for (int i = start; i < end; i++) }
        buffer.put (myString.charAt (i));
}
```

对 String 要进行与 char 数组相同的范围检查。如果所有的字符都不适合缓冲区，将会抛出 BufferOverflowException 异常。

2.2 创建缓冲区

就像我们在图 2-1 所看到的那样，有七种主要的缓冲区类，每一种都具有一种 Java 语言中的非布尔类型的原始类型数据。（第 8 种也在图中显示出来，MappedByteBuffer，是 ByteBuffer 专门用于内存映射文件的一种特例。我们将会在第三章讨论内存映射）。这些类没有一种能够直接实例化。它们都是抽象类，但是都包含静态工厂方法用来创建相应类的新实例。

对于这一讨论，我们将以 CharBuffer 类为例，但是对于其它六种主要的缓冲区类也是适用的：IntBuffer，DoubleBuffer，ShortBuffer，LongBuffer，FloatBuffer，和 ByteBuffer。下面是创建一个缓冲区的关键函数，对所有的缓冲区类通用（要按照需要替换类名）：

```
public abstract class CharBuffer
        extends Buffer implements CharSequence, Comparable
{
        // This is a partial API listing

        public static CharBuffer allocate (int capacity)

        public static CharBuffer wrap (char [] array)
        public static CharBuffer wrap (char [] array, int offset,
  int length)

        public final boolean hasArray(  )
        public final char [] array(  )
        public final int arrayOffset(  )
}
```

新的缓冲区是由分配或包装操作创建的。分配操作创建一个缓冲区对象并分配一个私有的空间来储存容量大小的数据元素。包装操作创建一个缓冲区对象但是不分配任何空间来储存数据元素。它使用您所提供的数组作为存储空间来储存缓冲区中的数据元素。

要分配一个容量为 100 个 char 变量的 Charbuffer:

```
CharBuffer charBuffer = CharBuffer.allocate (100);
```

这段代码隐含地从堆空间中分配了一个 char 型数组作为备份存储器来储存 100 个 char 变量。

如果您想提供您自己的数组用做缓冲区的备份存储器，请调用 wrap()函数：

```
char [] myArray = new char [100];
CharBuffer charbuffer = CharBuffer.wrap (myArray);
```

这段代码构造了一个新的缓冲区对象，但数据元素会存在于数组中。这意味着通过调用 put()函数造成的对缓冲区的改动会直接影响这个数组，而且对这个数组的任何改动也会对这个缓冲区对象可见。带有 offset 和 length 作为参数的 wrap()函数版本则会构造一个按照您提供的 offset 和 length 参数值初始化位置和上界的缓冲区。这样做：

```
CharBuffer charbuffer = CharBuffer.wrap (myArray, 12, 42);
```

创建了一个 position 值为 12，limit 值为 54，容量为 myArray.length 的缓冲区。

这个函数并不像您可能认为的那样，创建了一个只占用了一个数组子集的缓冲区。这个缓冲区可以存取这个数组的全部范围；offset 和 length 参数只是设置了初始的状态。调用使用上面代码中的方法创建的缓冲区中的 clear()函数，然后对其进行填充，直到超过上界值，这将会重写数组中的所有元素。Slice()函数（2.3 节将会讨论）可以提供一个只占用备份数组一部分的缓冲区。

通过 allocate()或者 wrap()函数创建的缓冲区通常都是间接的（直接缓冲区会在 2.4.2 节讨论）。间接的缓冲区使用备份数组，像我们之前讨论的，您可以通过上面列出的 API 函数获得对这些数组的存取权。Boolean 型函数 hasArray()告诉您这个缓冲区是否有一个可存取的备份数组。如果这个函数的返回 true，array()函数会返回这个缓冲区对象所使用的数组存储空间的引用。

如果 hasArray()函数返回 false，不要调用 array()函数或者 arrayOffset()函数。如果您这样做了您会得到一个 UnsupportedOperationException 异常。如果一个缓冲区是只读的，它的备份数组将会是超出上界的，即使一个数组对象被提供给 wrap()函数。调用 array()函数或者 arrayOffset()会抛出一个 ReadOnlyBufferException 异常，来阻止您得到存取权来修改只读缓冲区的内容。如果您通过其它的方式获得了对备份数组的存取权限，对这个数组的修改也会直接影响到这个只读缓冲区。只读缓冲区将会在 2.3 节讨论。

最后一个函数，arrayOffset()，返回缓冲区数据在数组中存储的开始位置的偏移量（从数组头 0 开始计算）。如果您使用了带有三个参数的版本的 wrap()函数来创建一个缓冲区，对于这个缓冲区，arrayOffset()会一直返回 0，像我们之前讨论的那样。然而，如果您切分了由一个数组提供存储的缓冲区，得到的缓冲区可能会有一个非 0 的数组偏移量。这个数组偏移量和缓冲区容量值会告诉您数组中哪些元素是被缓冲区使用的。缓冲区的切分会在 2.3 节讨论。

到现在为止，这一节所进行的讨论已经针对了所有的缓冲区类型。我们用来做例子的 CharBuffer 提供了一对其它缓冲区类没有的有用的便捷的函数：

```
public abstract class CharBuffer
       extends Buffer implements CharSequence, Comparable
{
       // This is a partial API listing
```

```
        public static CharBuffer wrap (CharSequence csq)
        public static CharBuffer wrap (CharSequence csq, int start,
  int end)
 }
```

Wrap()函数创建一个只读的备份存储区是 CharSequence 接口或者其实现的的缓冲区对象。（CharSequence 对象将在第五章讨论）。Charsequence 描述了一个可读的字符流。像 JDK1.4 中，三个标准的实现了 Charsequence 接口的类：String，StringBuffer，和 CharBuffer。Wrap()函数可以很实用地“缓冲”一个已有的字符数据，通过缓冲区的 API 来存取其中的内容。对于字符集解码（第六章）和正则表达式处理（第五章）这将是非常方便的。

```
CharBuffer charBuffer = CharBuffer.wrap ("Hello World");
```

三个参数的 wrap()函数版本使用 start 和 end 下标参数来描述传入的 CharSequence 对象的子序列。这是一个方便的类似于调用了 CharSequence.subsequence()函数的转换。Start 参数是序列中使用的第一个字符，end 是最后一个字符的下标值加 1。

2.3 复制缓冲区

如我们刚刚所讨论的那样，可以创建描述从外部存储到数组中的数据元素的缓冲区对象。但是缓冲区不限于管理数组中的外部数据。它们也能管理*其他缓冲区*中的外部数据。当一个管理其他缓冲器所包含的数据元素的缓冲器被创建时，这个缓冲器被称为视图缓冲器。大多数的视图缓冲器都是 ByteBuffer（参见 2.4.3 节）的视图。在继续前往字节缓冲器的细节之前，我们先将注意力放在所有存储器类型的共同视图上。

视图存储器总是通过调用已存在的存储器实例中的函数来创建。使用已存在的存储器实例中的工厂方法意味着视图对象为原始存储器的内部实现细节私有。数据元素可以直接存取，无论它们是存储在数组中还是以一些其他的方式，而不需经过原始缓冲区对象的 get()/put() API。如果原始缓冲区是直接缓冲区，该缓冲区的视图会具有同样的效率优势。映像缓冲区也是如此（将于第三章讨论）。

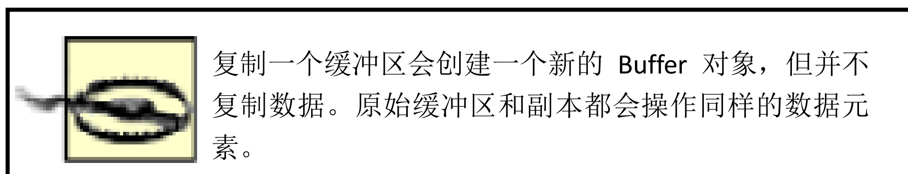
在这一章节中，我们将再次以 CharBuffer 为例，但同样的操作可被用于任何基本的缓冲区类型（参见图 2.1）。

```
public abstract class CharBuffer
        extends Buffer implements CharSequence, Comparable
{
        // This is a partial API listing

        public abstract CharBuffer duplicate( );
        public abstract CharBuffer asReadOnlyBuffer( );
        public abstract CharBuffer slice( );
}
```

Duplicate()函数创建了一个与原始缓冲区相似的新缓冲区。两个缓冲区共享数据元

素，拥有同样的容量，但每个缓冲区拥有各自的位置，上界和标记属性。对一个缓冲区内的数据元素所做的改变会反映在另外一个缓冲区上。这一副本缓冲区具有与原始缓冲区同样的数据视图。如果原始的缓冲区为只读，或者为直接缓冲区，新的缓冲区将继承这些属性。直接缓冲区将在 2.4.2 节中讨论。

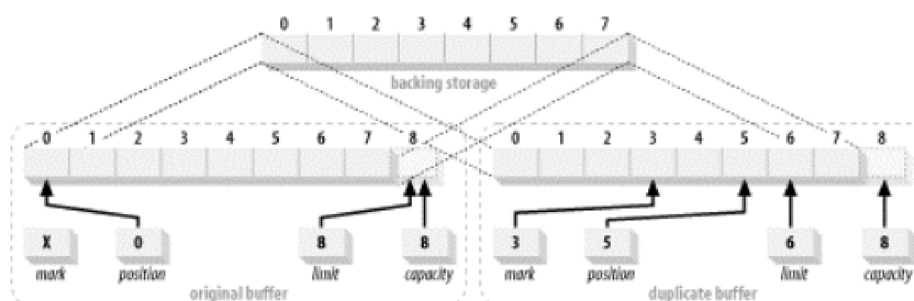


复制一个缓冲区会创建一个新的 Buffer 对象，但并不复制数据。原始缓冲区和副本都会操作同样的数据元素。

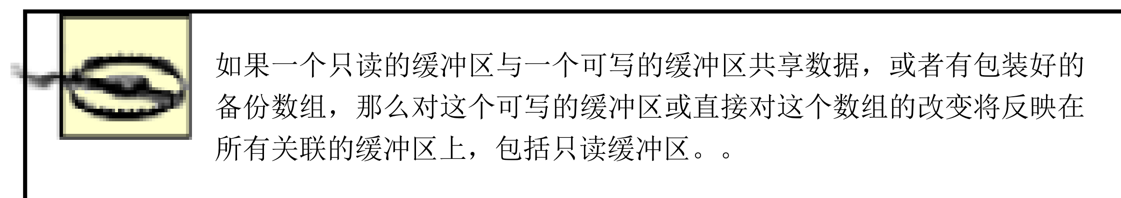
缓冲区及其副本之间的联系如图 2.12 所示。这是如下文所示的代码产生的：

```
CharBuffer buffer = CharBuffer.allocate (8);
buffer.position (3).limit (6).mark( ).position (5);
CharBuffer dupeBuffer = buffer.duplicate( );
buffer.clear( );
```

图 2-12 复制一个缓冲区



您可以使用 asReadOnlyBuffer()函数来生成一个只读的缓冲区视图。这与 duplicate()相同，除了这个新的缓冲区不允许使用 put()，并且其 isReadOnly()函数将会返回 true。对这一只读缓冲区的 put()函数的调用尝试会导致抛出 ReadOnlyBufferException 异常。

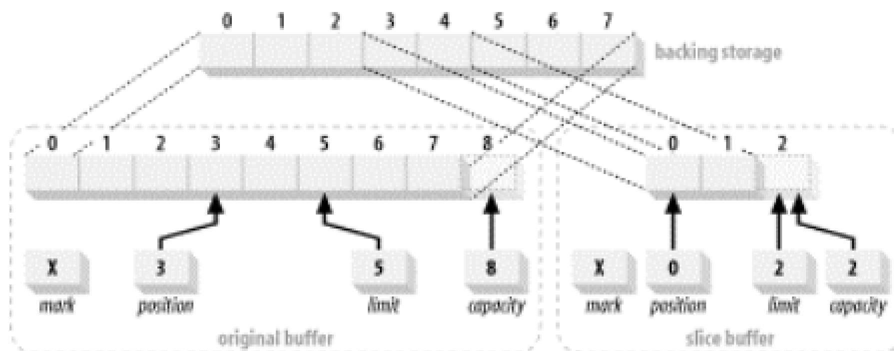


如果一个只读的缓冲区与一个可写的缓冲区共享数据，或者有包装好的备份数组，那么对这个可写的缓冲区或直接对这个数组的改变将反映在所有关联的缓冲区上，包括只读缓冲区。。

分割缓冲区与复制相似，但 slice()创建一个从原始缓冲区的当前位置开始的新缓冲区，并且其容量是原始缓冲区的剩余元素数量（limit-position）。这个新缓冲区与原始缓冲区共享一段数据元素子序列。分割出来的缓冲区也会继承只读和直接属性。图 2-13 显示了以与下面代码相似的代码所生成的分割缓冲区：

```
CharBuffer buffer = CharBuffer.allocate (8);
buffer.position (3).limit (5);
CharBuffer sliceBuffer = buffer.slice( );
```

图 2-13 创建分割缓冲区



要创建一个映射到数组位置 12-20（9 个元素）的 buffer 对象，应使用下面的代码实现：

```
char [] myBuffer = new char [100];
CharBuffer cb = CharBuffer.wrap (myBuffer);
cb.position(12).limit(21);
CharBuffer sliced = cb.slice(  );
```

更详细关于视图 buffer 的讨论参见 2.4.3 节。

2.4 字节缓冲区

在本章节中，我们将进一步观察字节缓冲区。所有的基本数据类型都有相应的缓冲区类（布尔型除外），但字节缓冲区有自己的独特之处。字节是操作系统及其 I/O 设备使用的基本数据类型。当在 JVM 和操作系统间传递数据时，将其他的数据类型拆分成构成它们的字节是十分必要的。如我们在后面的章节中将要看到的那样，系统层次的 I/O 面向字节的性质可以在整个缓冲区的设计以及它们互相配合的服务中感受到。

为了提供参考，以下是 ByteBuffer 的完整 API。这些函数有些已经在前面的章节中讨论，并且仅仅是针对具体类型的版本。新的函数将在本节以及后面的章节中涉及。

```
package java.nio;

public abstract class ByteBuffer extends Buffer
       implements Comparable
{
       public static ByteBuffer allocate (int capacity)
       public static ByteBuffer allocateDirect (int capacity)
       public abstract boolean isDirect(  );
       public static ByteBuffer wrap (byte[] array, int offset, int length)
       public static ByteBuffer wrap (byte[] array)

       public abstract ByteBuffer duplicate(  );
```

```
}
        public abstract ByteBuffer asReadOnlyBuffer( );
        public abstract ByteBuffer slice( );
        public final boolean hasArray( )
        public final byte [] array( )
        public final int arrayOffset( )

        public abstract byte get( );
        public abstract byte get (int index);
        public ByteBuffer get (byte[] dst, int offset, int length)
        public ByteBuffer get (byte[] dst, int offset, int length)

        public abstract ByteBuffer put (byte b);
        public abstract ByteBuffer put (int index, byte b);
        public ByteBuffer put (ByteBuffer src)
        public ByteBuffer put (byte[] src, int offset, int length)
        public final ByteBuffer put (byte[] src)

        public final ByteOrder order( )
        public final ByteBuffer order (ByteOrder bo)

        public abstract CharBuffer asCharBuffer( );
        public abstract ShortBuffer asShortBuffer( );
        public abstract IntBuffer asIntBuffer( );
        public abstract LongBuffer asLongBuffer( );
        public abstract FloatBuffer asFloatBuffer( );
        public abstract DoubleBuffer asDoubleBuffer( );

        public abstract char getChar( );
        public abstract char getChar (int index);
        public abstract ByteBuffer putChar (char value);
        public abstract ByteBuffer putChar (int index, char value);

        public abstract short getShort( );
        public abstract short getShort (int index);
        public abstract ByteBuffer putShort (short value);
        public abstract ByteBuffer putShort (int index, short value);

        public abstract int getInt( );
        public abstract int getInt (int index);
        public abstract ByteBuffer putInt (int value);
        public abstract ByteBuffer putInt (int index, int value);

        public abstract long getLong( );
        public abstract long getLong (int index);
        public abstract ByteBuffer putLong (long value);
        public abstract ByteBuffer putLong (int index, long value);

        public abstract float getFloat( );
        public abstract float getFloat (int index);
        public abstract ByteBuffer putFloat (float value);
         public abstract ByteBuffer putFloat (int index, float value);

         public abstract double getDouble( );
         public abstract double getDouble (int index);
         public abstract ByteBuffer putDouble (double value);
         public abstract ByteBuffer putDouble (int index, double value);

         public abstract ByteBuffer compact( );
         public boolean equals (Object ob) {
         public int compareTo (Object ob) {
         public String toString( )
         public int hashCode( )
}
```

字节总是八位的，对吗？

目前，字节几乎被广泛认为是八个比特位。但这并非一直是实情。在过去的时代，每个字节可以是 3 到 12 之间任何个数或者更多个的比特位，最常见的是 6 到 9 位。八位的字节来自于市场力量和实践的结合。它之所以实用是因为 8 位足以表达可用的字符集（至少是英文字符），8 是 2 的三次乘方（这简化了硬件设计），八恰好容纳两个十六进制数字，而且 8 的倍数提供了足够的组合位来存储有效的数值。市场力量是 IBM。在 1960 年首先推出的 IBM360 大型机使用的就是 8 位字节。这已经足够解决这个问题了。要想知道更多的背景，您可以去咨询 IBM 的 鲍 勃 • 贝 莫 本 人 ， 他 的 网 站 是 ：

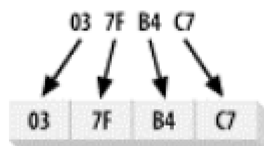
2.4.1 字节顺序

非字节类型的基本类型，除了布尔型[3]都是由组合在一起的几个字节组成的。这些数据类型及其大小总结在表 2-1 中。

表 2-1.基本数据类型及其大小	
数据类型	大小（以字节表示）
Byte	1
Char	2
Short	2
Int	4
Long	8
Float	4
Double	8

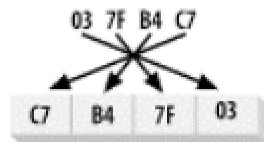
每个基本数据类型都是以连续字节序列的形式存储在内存中。例如，32 位的 int 值 0x037fb4c7（十进制的 58,700,999）可能会如图 2-14 所显示的那样被塞入内存字节中（内存地址从左往右增加）。注意前一个句子中的“可能”一词。尽管字节大小已经被确定，但字节顺序问题一直没有被广泛认同。表示一个整型值的字节可能在内存中仅仅如图 2-15 所示的那样被简单地排列。

图 2.14 大端字节顺序



[3] 布尔型代表两种值：true 或 false。一个字节可以包含 256 个唯一值，所以一个布尔值不能准确地对应到一个或多个字节上。字节是所有缓冲区的构造块。NIO 架构决定了布尔缓冲区的执行是有疑问的，而且对这种缓冲区类型的需求不管怎么说都是颇具争议的。

图 2.15 小端字节顺序



多字节数值被存储在内存中的方式一般被称为 endian-ness（字节顺序）。如果数字数值的最高字节——big end（大端），位于低位地址，那么系统就是大端字节顺序（如图 2-14 所示）。如果最低字节最先保存在内存中，那么小端字节顺序（如图 2-15 所示）。

字节顺序很少由软件设计者决定；它通常取决于硬件设计。字节顺序的两种类型有时被称为字节性别，在当今被广泛使用。两种方式都具有自身的优势。Intel 处理器使用小端字节顺序涉及。摩托罗拉的 CPU 系列、SUN 的 Sparc 工作站，以及 PowerPC 的 CPU 架构都采用大端字节顺序。

字节顺序的问题甚至胜过CPU硬件设计。当Internet的设计者为互联各种类型的计算机而设计网际协议（IP）时，他们意识到了在具有不同内部字节顺序的系统间传递数值数据的问题。因此，IP协议规定了使用大端的网络字节顺序概念 4。所有在IP分组报文的协议部分中使用的多字节数值必须先在本地主机字节顺序和通用的网络字节顺序之间进行转换。

在 java.nio 中，字节顺序由 ByteOrder 类封装。

```
package java.nio;

public final class ByteOrder
{
      public static final ByteOrder BIG_ENDIAN
      public static final ByteOrder LITTLE_ENDIAN

      public static ByteOrder nativeOrder( )
      public String toString( )
}
```

ByteOrder 类定义了决定从缓冲区中存储或检索多字节数值时使用哪一字节顺序的常量。这个类的作用就像一个类型安全的枚举。它定义了以其本身实例预初始化的两个 public 区域。只有这两个 ByteOrder 实例总是存在于 JVM 中，因此它们可以通过使用--操作符进行比较。如果您需要知道 JVM 运行的硬件平台的固有字节顺序，请调用静态类函数 nativeOrder()。它将返回两个已确定常量中的一个。调用 toString()将返回一个包含两个文字字符串 BIG_ENDIAN 或者 LITTLE_ENDIAN 之一的 String。

每个缓冲区类都具有一个能够通过调用 order()查询的当前字节顺序设定。

```
public abstract class CharBuffer extends Buffer
      implements Comparable, CharSequence
{
      // This is a partial API listing

      public final ByteOrder order( )
}
```

这个函数从 ByteOrder 返回两个常量之一。对于除了 ByteOrder 之外的其他缓冲区类，字节顺序是一个只读属性，并且可能根据缓冲区的建立方式而采用不同的值。除了

---

4 因特网技术将字节成为八位位组（octet）。如侧栏所介绍的，一个字节的大小是有歧义的。通过使用“octet”这一术语，IP 规则明确表明字节由八个位组成。

ByteBuffer，其他通过分配或包装一个数组所创建的缓冲区将从 order()返回与 ByteOrder.nativeOrder()相同的数值。这使因为包含在缓冲区中的元素在 JVM 中将会被作为基本数据直接存取。

ByteBuffer 类有所不同：默认字节顺序总是 ByteBuffer.BIG_ENDIAN，无论系统的固有字节顺序是什么。Java 的默认字节顺序是大端字节顺序，这允许类文件等以及串行化的对象可以在任何 JVM 中工作。如果固有硬件字节顺序是小端，这会有性能隐患。在使用固有硬件字节顺序时，将 ByteBuffer 的内容当作其他数据类型存取（很快就会讨论到）很可能高效得多。

很可能您会对为什么 ByteBuffer 类需要一个字节顺序设定这一问题感到困惑。字节就是字节，对吗？当然，但是如您不久将在 2.4.4 节所看到的那样，ByteBuffer 对象像其他基本数据类型一样，具有大量便利的函数用于获取和存放缓冲区内容。这些函数对字节进行编码或解码的方式取决于 ByteBuffer 当前字节顺序的设定。

ByteBuffer 的字符顺序设定可以随时通过调用以 ByteOrder.BIG_ENDIAN 或 ByteOrder.LITTL_ENDIAN 为参数的 order()函数来改变。

```
public abstract class ByteBuffer extends Buffer
        implements Comparable
{
        // This is a partial API listing

        public final ByteOrder order(  )
        public final ByteBuffer order (ByteOrder bo)
}
```

如果一个缓冲区被创建为一个 ByteBuffer 对象的视图（参见 2.4.3 节），那么 order()返回的数值就是视图被创建时其创建源头的 ByteBuffer 的字节顺序设定。视图的字节顺序设定在创建后不能被改变，而且如果原始的字节缓冲区的字节顺序在之后被改变，它也不会受到影响。。

2.4.2 直接缓冲区

字节缓冲区跟其他缓冲区类型最明显的不同在于，它们可以成为通道所执行的 I/O 的源头和/或目标。如果您向前跳到第三章（喂！喂！），您会发现通道只接收 ByteBuffer 作为参数。

如我们在第一章中所看到的那样，操作系统的在内存区域中进行 I/O 操作。这些内存区域，就操作系统方面而言，是相连的字节序列。于是，毫无疑问，只有字节缓冲区有资格参与 I/O 操作。也请回想一下操作系统会直接存取进程——在本例中是 JVM 进程的内存空间，以传输数据。这也意味着 I/O 操作的目标内存区域必须是连续的字节序列。在 JVM 中，字节数组可能不会在内存中连续存储，或者无用存储单元收集可能随时对其进行移动。在 Java 中，数组是对象，而数据存储在对象中的方式在不同的 JVM 实现中都各有不同。

出于这一原因，引入了直接缓冲区的概念。直接缓冲区被用于与通道和固有 I/O 例程交互。它们通过使用固有代码来告知操作系统直接释放或填充内存区域，对用于通道直接或原始存取的内存区域中的字节元素的存储尽了最大的努力。

直接字节缓冲区通常是 I/O 操作最好的选择。在设计方面，它们支持 JVM 可用的最高效 I/O 机制。非直接字节缓冲区可以被传递给通道，但是这样可能导致性能损耗。通常非直接缓冲不可能成为一个本地 I/O 操作的目标。如果您向一个通道中传递一个非直接 ByteBuffer 对象用于写入，通道可能会在每次调用中隐含地进行下面的操作：

1. 创建一个临时的直接 ByteBuffer 对象。
2. 将非直接缓冲区的内容复制到临时缓冲中。
3. 使用临时缓冲区执行低层次 I/O 操作。
4. 临时缓冲区对象离开作用域，并最终成为被回收的无用数据。

这可能导致缓冲区在每个 I/O 上复制并产生大量对象，而这种事都是我们极力避免的。不过，依靠工具，事情可以不这么糟糕。运行时间可能会缓存并重新使用直接缓冲区或者执行其他一些聪明的技巧来提高吞吐量。如果您仅仅为一次使用而创建了一个缓冲区，区别并不是很明显。另一方面，如果您将在一段高性能脚本中重复使用缓冲区，分配直接缓冲区并重新使用它们会使您游刃有余。

直接缓冲区时 I/O 的最佳选择，但可能比创建非直接缓冲区要花费更高的成本。直接缓冲区使用的内存是通过调用本地操作系统方面的代码分配的，绕过了标准 JVM 堆栈。建立和销毁直接缓冲区会明显比具有堆栈的缓冲区更加破费，这取决于主操作系统以及 JVM 实现。直接缓冲区的内存区域不受无用存储单元收集支配，因为它们位于标准 JVM 堆栈之外。

使用直接缓冲区或非直接缓冲区的性能权衡会因JVM，操作系统，以及代码设计而产生巨大差异。通过分配堆栈外的内存，您可以使您的应用程序依赖于JVM未涉及的其它力量。当加入其他的移动部分时，确定您正在达到想要的效果。我以一条旧的软件行业格言建议您：先使其工作，再加快其运行。不要一开始就过多担心优化问题；首先要注重正确性。JVM实现可能会执行缓冲区缓存或其他的优化，[5]这会在不需要您参与许多不必要工作的情况下为您提供所需的性能。

直接 ByteBuffer 是通过调用具有所需容量的 ByteBuffer.allocateDirect()函数产生的，就像我们之前所涉及的 allocate()函数一样。注意用一个 wrap()函数所创建的被包装的缓冲区总是非直接的。

```
public abstract class ByteBuffer
       extends Buffer implements Comparable
{
       // This is a partial API listing

       public static ByteBuffer allocate (int capacity)
       public static ByteBuffer allocateDirect (int capacity)
       public abstract boolean isDirect(  );
}
```

所有的缓冲区都提供了一个叫做 isDirect()的 boolean 函数，来测试特定缓冲区是否为直接缓冲区。虽然 ByteBuffer 是唯一可以被直接分配的类型，但如果基础缓冲区是一个直接 ByteBuffer，对于非字节视图缓冲区，isDirect()可以是 true。这将我们带到了……

---

[5] “我们应该忘记小的效率，因为97%的情况下：过早的优化是所有祸害的根源。”——唐纳德·克努特。

2.4.3 视图缓冲区

就像我们已经讨论的那样，I/O 基本上可以归结成组字节数据的四处传递。在进行大数据量的 I/O 操作时，很又可能您会使用各种 ByteBuffer 类去读取文件内容，接收来自网络连接的数据，等等。一旦数据到达了您的 ByteBuffer，您就需要查看它以决定怎么做或者在将它发送出去之前对它进行一些操作。ByteBuffer 类提供了丰富的 API 来创建视图缓冲区。

视图缓冲区通过已存在的缓冲区对象实例的工厂方法来创建。这种视图对象维护它自己的属性，容量，位置，上界和标记，但是和原来的缓冲区共享数据元素。我们已经在 2.3 节见过了这样的简单例子，在例子中一个缓冲区被复制和切分。但是 ByteBuffer 类允许创建视图来将 byte 型缓冲区字节数据映射为其它的原始数据类型。例如，asLongBuffer()函数创建一个将八个字节型数据当成一个 long 型数据来存取的视图缓冲区。

下面列出的每一个工厂方法都在原有的 ByteBuffer 对象上创建一个视图缓冲区。调用其中的任何一个方法都会创建对应的缓冲区类型，这个缓冲区是基础缓冲区的一个切分，由基础缓冲区的位置和上界决定。新的缓冲区的容量是字节缓冲区中存在的元素数量除以视图类型中组成一个数据类型的字节数（参见表 2-1）。在切分中任一个超过上界的元素对于这个视图缓冲区都是不可见的。视图缓冲区的第一个元素从创建它的 ByteBuffer 对象的位置开始（positon()函数的返回值）。具有能被自然数整除的数据元素个数的视图缓冲区是一种较好的实现。

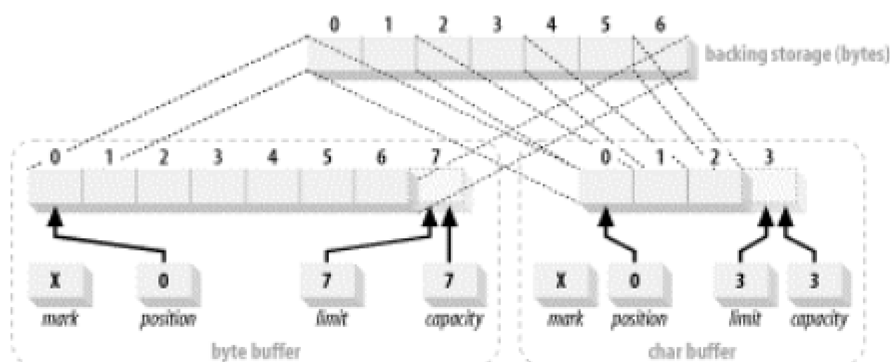
```
public abstract class ByteBuffer
       extends Buffer implements Comparable
{
       // This is a partial API listing

       public abstract CharBuffer asCharBuffer( );
       public abstract ShortBuffer asShortBuffer( );
       public abstract IntBuffer asIntBuffer( );
       public abstract LongBuffer asLongBuffer( );
       public abstract FloatBuffer asFloatBuffer( );
       public abstract DoubleBuffer asDoubleBuffer( );
  }
```

下面的代码创建了一个 ByteBuffer 缓冲区的 CharBuffer 视图，如图 Figure 2-16 所示（Example 2-2 将这个框架用到了更大的范围）

```
ByteBuffer byteBuffer =
 ByteBuffer.allocate (7).order (ByteOrder.BIG_ENDIAN);
CharBuffer charBuffer = byteBuffer.asCharBuffer( );
```

图 2-16. 一个 ByteBuffer 的 CharBuffer 视图



例 2-2. 创建一个 ByteBuffer 的字符视图

```
package com.ronsoft.books.nio.buffers;

import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.ByteOrder;

/**
 * Test asCharBuffer view.
 *
 * Created May 2002
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class BufferCharView
{
   public static void main (String [] argv)
      throws Exception
   {
      ByteBuffer byteBuffer =
         ByteBuffer.allocate (7).order (ByteOrder.BIG_ENDIAN);
      CharBuffer charBuffer = byteBuffer.asCharBuffer(  );

      // Load the ByteBuffer with some bytes
      byteBuffer.put (0, (byte)0);
      byteBuffer.put (1, (byte)'H');
      byteBuffer.put (2, (byte)0);
      byteBuffer.put (3, (byte)'i');
      byteBuffer.put (4, (byte)0);
      byteBuffer.put (5, (byte)'!');
      byteBuffer.put (6, (byte)0);

      println (byteBuffer);
      println (charBuffer);
   }

   // Print info about a buffer
   private static void println (Buffer buffer)
   {
      System.out.println ("pos=" + buffer.position(  )
         + ", limit=" + buffer.limit(  )
         + ", capacity=" + buffer.capacity(  )
         + ": '" + buffer.toString(  ) + "'");
   }
}
pos=0, limit=7, capacity=7: 'java.nio.HeapByteBuffer[pos=0 lim=7 cap=7]'
pos=0, limit=3, capacity=3: 'Hi!'
```

运行 BufferCharView 程序的输出是：

```
pos=0, limit=7, capacity=7: 'java.nio.HeapByteBuffer[pos=0 lim=7 cap=7]'
pos=0, limit=3, capacity=3: 'Hi!
```

一旦您得到了视图缓冲区，您可以用 duplicate()，slice() 和 asReadOnlyBuffer()函数创建进一步的子视图，就像 2.3 节所讨论的那样。

无论何时一个视图缓冲区存取一个 ByteBuffer 的基础字节，这些字节都会根据这个视图缓冲区的字节顺序设定被包装成一个数据元素。当一个视图缓冲区被创建时，视图创建的同时它也继承了基础 ByteBuffer 对象的字节顺序设定。这个视图的字节排序不能再被修改。在图 2-16 中，您可以看到基础 ByteBuffer 对象中的两个字节映射成 CharBuffer 对象中的一个字符。字节顺序设定决定了这些字节对是怎么样被组合成字符型变量的。请参考 2.4.1 节获取更多详细的解释。

当直接从 byte 型缓冲区中采集数据时，视图换冲突拥有提高效率的潜能。如果这个视图的字节顺序和本地机器硬件的字节顺序一致，低等级的（相对于高级语言而言）语言的代码可以直接存取缓冲区中的数据值，而不是通过比特数据的包装和解包装过程来完成。

2.4.4 数据元素视图

ByteBuffer 类提供了一个不太重要的机制来以多字节数据类型的形式存取 byte 数据组。ByteBuffer 类为每一种原始数据类型提供了存取的和转化的方法：

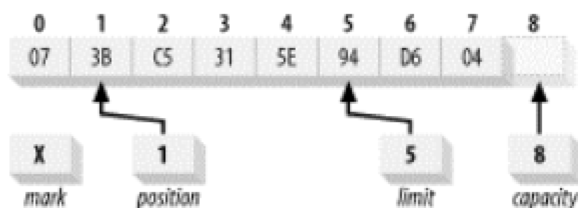
```
public abstract class ByteBuffer
       extends Buffer implements Comparable
{
       public abstract char getChar(  );
       public abstract char getChar (int index);
       public abstract short getShort(  );
       public abstract short getShort (int index);
       public abstract int getInt(  );
       public abstract int getInt (int index);
       public abstract long getLong(  );
       public abstract long getLong (int index);
       public abstract float getFloat(  );
       public abstract float getFloat (int index);
       public abstract double getDouble(  );
       public abstract double getDouble (int index);

       public abstract ByteBuffer putChar (char value);
       public abstract ByteBuffer putChar (int index, char value);
       public abstract ByteBuffer putShort (short value);
       public abstract ByteBuffer putShort (int index, short value);
       public abstract ByteBuffer putInt (int value);
       public abstract ByteBuffer putInt (int index, int value);
       public abstract ByteBuffer putLong (long value);
       public abstract ByteBuffer putLong (int index, long value);
       public abstract ByteBuffer putFloat (float value);
       public abstract ByteBuffer putFloat (int index, float value);
       public abstract ByteBuffer putDouble (double value);
       public abstract ByteBuffer putDouble (int index, double value);
}
```

这些函数从当前位置开始存取 ByteBuffer 的字节数据，就好像一个数据元素被存储在那里一样。根据这个缓冲区的当前的有效的字节顺序，这些字节数据会被排列或打乱成需要的原始数据类型。比如说，如果 getInt()函数被调用，从当前的位置开始的四个字节会被包装成一个 int 类型的变量然后作为函数的返回值返回。参见 2.4.1 节。

假设一个叫 buffer 的 ByteBuffer 对象处于图 Figure 2-17 的状态。

图 2-17. 包含一些数据的 ByteBuffer



这段代码：

```
int value = buffer.getInt( );
```

会返回一个由缓冲区中位置 1-4 的 byte 数据值组成的 int 型变量的值。实际的返回值取决于缓冲区的当前的比特排序（byte-order）设置。更具体的写法是：

```
int value = buffer.order (ByteOrder.BIG_ENDIAN).getInt( );
```

这将会返回值 0x3BC5315E，同时：

```
int value = buffer.order (ByteOrder.LITTLE_ENDIAN).getInt( );
```

返回值 0x5E31C53B。

如果您试图获取的原始类型需要比缓冲区中存在的字节数更多的字节，会抛出 BufferUnderflowException。对于图 Figure2-17 中的缓冲区，这段代码会抛出异常，因为一个 long 型变量是 8 个字节的，但是缓冲区中只有 5 个字节：

```
long value = buffer.getLong( );
```

这些函数返回的元素不需要被任何特定模块界限所限制 [6]。数值将会从以缓冲区的当前位置开始的字节缓冲区中取出并组合，无论字组是否对齐。这样的做法是低效的，但是它允许对一个字节流中的数据进行随机的放置。对于从二进制文件数据或者包装数据成特定平台的格式或者导出到外部的系统，这将是非常有用的。

Put 函数提供与 get 相反的操作。原始数据的值会根据字节顺序被分拆成一个个 byte 数据。如果存储这些字节数据的空间不够，会抛出 BufferOverflowException。

每一个函数都有重载的和无参数的形式。重载的函数对位置属性加上特定的字节数。然后无参数的形式则不改变位置属性。

2.4.5 存取无符号数据

Java 编程语言对无符号数值并没有提供直接的支持（除了 char 类型）。但是在许多情况下您需要将无符号的信息转化成数据流或者文件，或者包装数据来创建文件头或者其它带有无符号数据区域结构化的信息。ByteBuffer 类的 API 对此并没有提供直接的支持，但是要实现并不困难。您只需要小心精度的问题。当您必须处理缓冲区中的无符号数据时，例 2-3 中的工具类可能会非常有帮助。

例 2-3. 获取/存放无符号值的工具程序

```
package com.ronsoft.books.nio.buffers;
```

[6] 模块考虑一个在用一个数除以另一个数时的余数。模块界限是对特定除数除数余数为零的数轴上的点。例如，任何能够被 4 均匀整除的数都是模块 4：4，8，12，16，等等。许多 CPU 设计要求模块内存的多字节数值排成直线。

```
import java.nio.ByteBuffer;
/**
* 向 ByteBuffer 对象中获取和存放无符号值的工具类。
* 这里所有的函数都是静态的，并且带有一个 ByteBuffer 参数。
* 由于 java 不提供无符号原始类型，每个从缓冲区中读出的无符号值被升到比它大的
* 下一个基本数据类型中。
* getUnsignedByte()返回一个 short 类型， getUnsignedShort( )

* 返回一个 int 类型，而 getUnsignedInt()返回一个 long 型。 There is no
* 由于没有基本类型来存储返回的数据，因此没有 getUnsignedLong( )。
* 如果需要，返回 BigInteger 的函数可以执行。
* 同样，存放函数要取一个大于它们所分配的类型的值。
* putUnsignedByte 取一个 short 型参数，等等。
*
* @author Ron Hitchens (ron@ronsoft.com)
*/


public class Unsigned
{
   public static short getUnsignedByte (ByteBuffer bb)
   {
      return ((short)(bb.get(  ) & 0xff));
   }

   public static void putUnsignedByte (ByteBuffer bb, int value)
   {
      bb.put ((byte)(value & 0xff));
   }

   public static short getUnsignedByte (ByteBuffer bb, int position)
   {
      return ((short)(bb.get (position) & (short)0xff));
   }

   public static void putUnsignedByte (ByteBuffer bb, int position,
      int value)
   {
      bb.put (position, (byte)(value & 0xff));
   }

   // ---------------------------------------------------------------

   public static int getUnsignedShort (ByteBuffer bb)
   {
      return (bb.getShort(  ) & 0xffff);
   }

   public static void putUnsignedShort (ByteBuffer bb, int value)
   {
      bb.putShort ((short)(value & 0xffff));
   }


   public static int getUnsignedShort (ByteBuffer bb, int position)
   {
      return (bb.getShort (position) & 0xffff);
   }

   public static void putUnsignedShort (ByteBuffer bb, int position,
      int value)
   {
      bb.putShort (position, (short)(value & 0xffff));
   }

   // ---------------------------------------------------------------

   public static long getUnsignedInt (ByteBuffer bb)
```

```
    {
       return ((long)bb.getInt(  ) & 0xffffffffL);
    }




    public static void putUnsignedInt (ByteBuffer bb, long value)
    {
       bb.putInt ((int)(value & 0xffffffffL));
    }

    public static long getUnsignedInt (ByteBuffer bb, int position)
    {
       return ((long)bb.getInt (position) & 0xffffffffL);
    }

    public static void putUnsignedInt (ByteBuffer bb, int position,
       long value)
    {
       bb.putInt (position, (int)(value & 0xffffffffL));
    }
}
```

2.4.6 内存映射缓冲区

映射缓冲区是带有存储在文件，通过内存映射来存取数据元素的字节缓冲区。映射缓冲区通常是直接存取内存的，只能通过 FileChannel 类创建。映射缓冲区的用法和直接缓冲区类似，但是 MappedByteBuffer 对象可以处理独立于文件存取形式的的许多特定字符。出于这个原因，我将讨论映射缓冲区的内容留到 3.4 节，在那里我们也会讨论文件锁。

2.5 总结

这一章介绍了 java.nio 包中的各种缓冲区。缓冲区对象使接下来几章要介绍的高吞吐率 I/O 得以实现。我们在这章讨论了以下关键内容：

缓冲区属性

所有缓冲区共有的属性在 2.1.1 节涉及。这些属性描述了缓冲区的当前状态，影响了缓冲区的表现。在这一节，我们也学习了怎样改变缓冲区的状态，以及如何增加及去除数据元素。

创建缓冲区

我们在 2.2 节学会了如何创建缓冲区，以及在 2.3 节学习了如何复制它们。缓冲区有许多类型。创建缓冲区的方式由缓冲区的使用方式和使用地点决定。

字节缓冲区

虽然缓冲区能够创建来缓冲除了布尔类型的原始数据类型数据，字节缓冲区却具有其他缓冲区类型没有的特征。只有字节缓冲区能够与通道共同使用（将于第三章讨论），并

且字节缓冲区提供了适用于其它数据类型的视图。我们也解释了字节排序的相关内容。

在 2.4 节我们讨论了 ByteBuffer 类。

以上总结了我们对 java.nio 包中的缓冲区小博物馆的拜访旅程。我们旅程的下一站是 java.nio.channel，在那里您毫无疑问会邂逅通道。通道和字节缓冲区进行交互，打开通往高性能 I/O 的大门。坐回我们的学习大巴吧，下一站很快就要到了。

第三章

通道

“辉煌！绝对的辉煌！”

—— *Wile E. Coyote* （*超级天才*）

通道（Channel）是 `java.nio` 的第二个主要创新。它们既不是一个扩展也不是一项增强，而是全新、极好的 Java I/O 示例，提供与 I/O 服务的直接连接。`Channel` 用于在字节缓冲区和位于通道另一侧的实体（通常是一个文件或套接字）之间有效地传输数据。

通道可以形象地比喻为银行出纳窗口使用的气动导管。您的薪水支票就是您要传送的信息，载体（Carrier）就好比一个缓冲区。您先填充缓冲区（将您的支票放到载体上），接着将缓冲“写”到通道中（将载体丢进导管中），然后信息负载就被传递到通道另一侧的 I/O 服务（银行出纳员）。

该过程的回应是：出纳员填充缓冲区（将您的收据放到载体上），接着开始一个反方向的通道传输（将载体丢回到导管中）。载体就到了通道的您这一侧（一个填满了的缓冲区正等待您的查验），然后您就会 flip 缓冲区（打开盖子）并将它清空（移除您的收据）。现在您可以开车走了，下一个对象（银行客户）将使用同样的载体（*Buffer*）和导管（*Channel*）对象来重复上述过程。

多数情况下，通道与操作系统的文件描述符（File Descriptor）和文件句柄（File Handle）有着一对一的关系。虽然通道比文件描述符更广义，但您将经常使用到的多数通道都是连接到开放的文件描述符的。Channel 类提供维持平台独立性所需的抽象过程，不过仍然会模拟现代操作系统本身的 I/O 性能。

通道是一种途径，借助该途径，可以用最小的总开销来访问操作系统本身的 I/O 服务。缓冲区则是通道内部用来发送和接收数据的端点。 （见图 3-1）

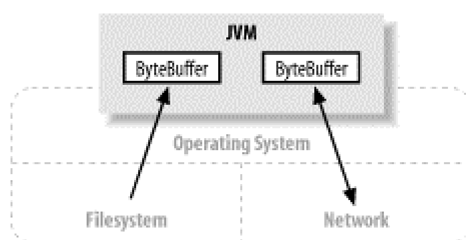


图 3-1 通道充当连接 I/O 服务的导管

观察图 3-2 所示的 UML 类图会发现，channel 类的继承关系要比 buffer 类复杂一些。Channel 类相互之间的关系更复杂，并且部分 channel 类依赖于在 `java.nio.channels.spi` 子包中定义

的类。本章我们将对该困惑进行澄清。通道 SPI 归纳参见附录 B。

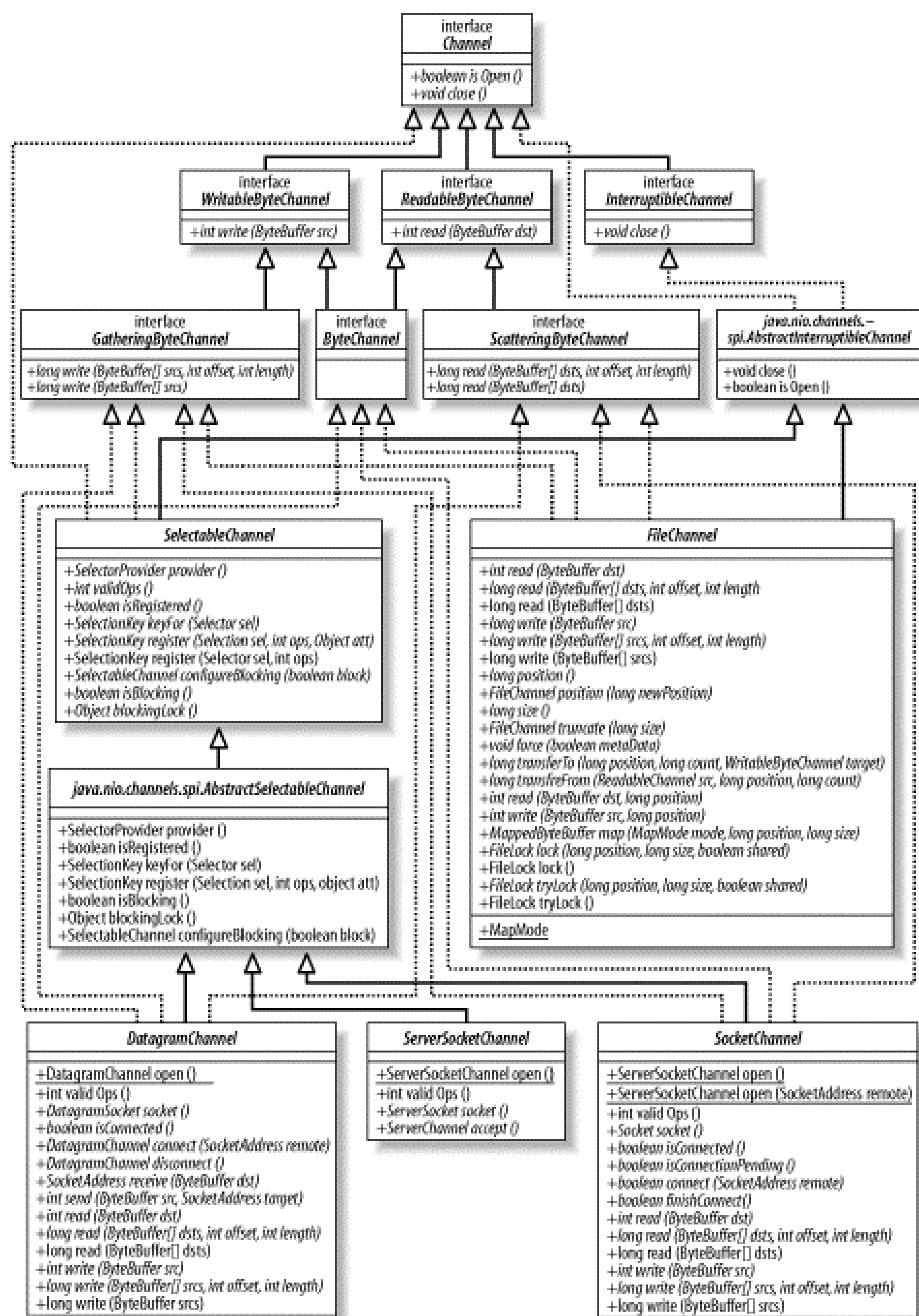


图 3-2 Channel 的类层次结构

不多讲了，让我们马上开始探索激动人心的通道世界吧！

3.1 通道基础

首先，我们来更近距离地看一下基本的 *Channel* 接口。下面是 *Channel* 接口的完整源码：

```
package java.nio.channels;
public interface Channel
{
    public boolean isOpen( );
    public void close( ) throws IOException;
}
```

与缓冲区不同，通道 API 主要由接口指定。不同的操作系统上通道实现（Channel Implementation）会有根本性的差异，所以通道 API 仅仅描述了可以做什么。因此很自然地，通道实现经常使用操作系统的本地代码。通道接口允许您以一种受控且可移植的方式来访问底层的 I/O 服务。

您可以从顶层的 Channel 接口看到，对所有通道来说只有两种共同的操作：检查一个通道是否打开（*IsOpen()*）和关闭一个打开的通道（*close()*）。图 3-2 显示，所有有趣的东西都是那些实现 *Channel* 接口以及它的子接口的类。

InterruptibleChannel 是一个标记接口，当被通道使用时可以标示该通道是可以中断的（Interruptible）。如果连接可中断通道的线程被中断，那么该通道会以特别的方式工作，关于这一点我们会在 3.1.3 节中进行讨论。大多数但非全部的通道都是可以中断的。

从 *Channel* 接口引申出的其他接口都是面向字节的子接口，包括 *Writable ByteChannel* 和 *ReadableByteChannel*。这也正好支持了我们之前所学的：通道只能在字节缓冲区上操作。层次结构表明其他数据类型的通道也可以从 *Channel* 接口引申而来。这是一种很好的类设计，不过非字节实现是不可能的，因为操作系统都是以字节的形式实现底层 I/O 接口的。

观察图 3-2，您还会发现类层次结构中有两个类位于一个不同的包：`java.nio.channels.spi`。这两个类是 *AbstractInterruptibleChannel* 和 *AbstractSelectableChannel*，它们分别为可中断的（interruptible）和可选择的（selectable）的通道实现提供所需的常用方法。尽管描述通道行为的接口都是在 `java.nio.channels` 包中定义的，不过具体的通道实现却都是从 `java.nio.channels.spi` 中的类引申来的。这使得他们可以访问受保护的方法，而这些方法普通的通道用户永远都不会调用。

作为通道的一个使用者，您可以放心地忽视 SPI 包中包含的中间类。这种有点费解的继承层次只会让那些使用新通道的用户感兴趣。SPI 包允许新通道实现以一种受控且模块化的方式被植入到

Java 虚拟机上。这意味着可以使用专为某种操作系统、文件系统或应用程序而优化的通道来使性能最大化。

3.1.1 打开通道

通道是访问 I/O 服务的导管。正如我们在第一章中所讨论的，I/O 可以分为广义的两大类别：File I/O 和 Stream I/O。那么相应地有两种类型的通道也就不足为怪了，它们是文件（file）通道和套接字（socket）通道。如果您参考一下图 3-2，您就会发现有一个 *FileChannel* 类和三个 socket 通道类：*SocketChannel*、*ServerSocketChannel* 和 *DatagramChannel*。

通道可以以多种方式创建。Socket 通道有可以直接创建新 socket 通道的工厂方法。但是一个 *FileChannel* 对象却只能通过在一个打开的 *RandomAccessFile*、*FileInputStream* 或 *FileOutputStream* 对象上调用 *getChannel( )*方法来获取。您不能直接创建一个 *FileChannel* 对象。File 和 socket 通道会在后面的章节中予以详细讨论。

```
SocketChannel sc = SocketChannel.open( );
sc.connect (new InetSocketAddress ("somehost", someport));
ServerSocketChannel ssc = ServerSocketChannel.open( );
ssc.socket( ).bind (new InetSocketAddress (somelocalport));
DatagramChannel dc = DatagramChannel.open( );
RandomAccessFile raf = new RandomAccessFile ("somefile", "r");
FileChannel fc = raf.getChannel( );
```

在 3.5 节中您会发现，`java.net` 的 socket 类也有新的 `getChannel( )`方法。这些方法虽然能返回一个相应的 socket 通道对象，但它们却并非新通道的来源，`RandomAccessFile.getChannel( )`方法才是。只有在已经有通道存在的时候，它们才返回与一个 socket 关联的通道；它们永远不会创建新通道。

3.1.2 使用通道

我们在第二章的学习中已经知道了，通道将数据传输给 *ByteBuffer* 对象或者从 *ByteBuffer* 对象获取数据进行传输。

将图 3-2 中大部分零乱内容移除可以得到图 3-3 所示的 UML 类图。子接口 API 代码如下：

```
public interface ReadableByteChannel
    extends Channel
{
    public int read (ByteBuffer dst) throws IOException;
```

```
}
public interface WritableByteChannel
    extends Channel
{
    public int write (ByteBuffer src) throws IOException;
}
public interface ByteChannel
    extends ReadableByteChannel, WritableByteChannel
{
}
```

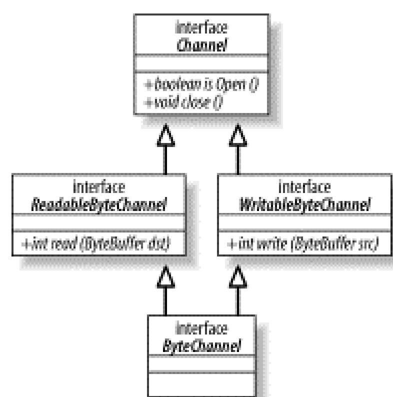


图 3-3 ByteChannel 接口

通道可以是单向（*unidirectional*）或者双向的（*bidirectional*）。一个 channel 类可能实现定义 *read( )*方法的 *ReadableByteChannel* 接口，而另一个 channel 类也许实现 *WritableByteChannel* 接口以提供 *write( )*方法。实现这两种接口其中之一的类都是单向的，只能在一个方向上传输数据。如果一个类同时实现这两个接口，那么它是双向的，可以双向传输数据。

图 3-3 显示了一个 *ByteChannel* 接口，该接口引申出了 *ReadableByteChannel* 和 *WritableByteChannel* 两个接口。*ByteChannel* 接口本身并不定义新的 API 方法，它是一种用来聚集它自己以一个新名称继承的多个接口的便捷接口。根据定义，实现 *ByteChannel* 接口的通道会同时实现 *ReadableByteChannel* 和 *WritableByteChannel* 两个接口，所以此类通道是双向的。这是简化类定义的语法糖（syntactic sugar），它使得用操作器（operator）实例来测试通道对象变得更加简单。

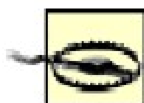
这是一种好的类设计技巧，如果您在写您自己的 Channel 实现的话，您可以适当地实现这些接口。不过对于使用 `java.nio.channels` 包中标准通道类的程序员来说，这些接口并没有太大的吸引力。假如您快速回顾一下图 3-2 或者向前跳跃到关于 file 和 socket 通道的章节，您将发现每一个 file 或 socket 通道都实现全部三个接口。从类定义的角度而言，这意味着全部 file 和 socket 通道对象都是双向的。这对于 sockets 不是问题，因为它们一直都是双向的，不过对于 files 却是个问题

了。

我们知道，一个文件可以在不同的时候以不同的权限打开。从 *FileInputStream* 对象的 *getChannel( )*方法获取的 *FileChannel* 对象是只读的，不过从接口声明的角度来看却是双向的，因为 *FileChannel* 实现 *ByteChannel* 接口。在这样一个通道上调用 *write( )*方法将抛出未经检查的 *NonWritableChannelException* 异常，因为 *FileInputStream* 对象总是以 read-only 的权限打开文件。

通道会连接一个特定 I/O 服务且通道实例（channel instance）的性能受它所连接的 I/O 服务的特征限制，记住这很重要。一个连接到只读文件的 *Channel* 实例不能进行写操作，即使该实例所属的类可能有 *write( )*方法。基于此，程序员需要知道通道是如何打开的，避免试图尝试一个底层 I/O 服务不允许的操作。

```
// A ByteBuffer named buffer contains data to be written
FileInputStream input = new FileInputStream (fileName);
FileChannel channel = input.getChannel( );
// This will compile but will throw an IOException
// because the underlying file is read-only
channel.write (buffer);
```



根据底层文件句柄的访问模式，通道实例可能不允许使用 *read()*或 *write()*方法。

ByteChannel 的 *read( )* 和 *write( )*方法使用 *ByteBuffer* 对象作为参数。两种方法均返回已传输的字节数，可能比缓冲区的字节数少甚至可能为零。缓冲区的位置也会发生与已传输字节相同数量的前移。如果只进行了部分传输，缓冲区可以被重新提交给通道并从上次中断的地方继续传输。该过程重复进行直到缓冲区的 *hasRemaining( )*方法返回 `false` 值。例 3-1 表示了如何从一个通道复制数据到另一个通道。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.IOException;
/**
  * Test copying between channels.
  *
```

```
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class ChannelCopy
{
    /**
       * This code copies data from stdin to stdout. Like the 'cat'
       * command, but without any useful options.
       */
    public static void main (String [] argv)
        throws IOException
    {
        ReadableByteChannel source = Channels.newChannel (System.in);
        WritableByteChannel dest = Channels.newChannel (System.out);
        channelCopy1 (source, dest);
        // alternatively, call channelCopy2 (source, dest);
        source.close( );
        dest.close( );
    }
    /**
      * Channel copy method 1. This method copies data from the src
      * channel and writes it to the dest channel until EOF on src.
      * This implementation makes use of compact( ) on the temp buffer
      * to pack down the data if the buffer wasn't fully drained. This
      * may result in data copying, but minimizes system calls. It also
      * requires a cleanup loop to make sure all the data gets sent.
      */
    private static void channelCopy1 (ReadableByteChannel src,
        WritableByteChannel dest)
        throws IOException
     {
        ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
        while (src.read (buffer) != -1) {
            // Prepare the buffer to be drained
            buffer.flip( );
            // Write to the channel; may block
            dest.write (buffer);
            // If partial transfer, shift remainder down
            // If buffer is empty, same as doing clear( )
```

```
            buffer.compact( );
        }
        // EOF will leave buffer in fill state
        buffer.flip( );
        // Make sure that the buffer is fully drained
        while (buffer.hasRemaining( )) {
            dest.write (buffer);
        }
    }
    /**
      * Channel copy method 2. This method performs the same copy, but
      * assures the temp buffer is empty before reading more data. This
      * never requires data copying but may result in more systems calls.
      * No post-loop cleanup is needed because the buffer will be empty
      * when the loop is exited.
      */
    private static void channelCopy2 (ReadableByteChannel src,
        WritableByteChannel dest)
        throws IOException
    {
        ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
        while (src.read (buffer) != -1) {
            // Prepare the buffer to be drained
            buffer.flip( );
            // Make sure that the buffer was fully drained
            while (buffer.hasRemaining( )) {
            dest.write (buffer);
            }
            // Make the buffer empty, ready for filling
            buffer.clear( );
        }
    }
}
```

例 3-1 在通道之间复制数据

通道可以以阻塞（*blocking*）或非阻塞（*nonblocking*）模式运行。非阻塞模式的通道永远不会让调用的线程休眠。请求的操作要么立即完成，要么返回一个结果表明未进行任何操作。只有面向流的（stream-oriented）的通道，如 sockets 和 pipes 才能使用非阻塞模式。

从图 3-2 可以看出，socket 通道类从 *SelectableChannel* 引申而来。从 *SelectableChannel* 引申而来的类可以和支持有条件的选择（readiness selectio）的选择器（Selectors）一起使用。将非阻塞 I/O 和选择器组合起来可以使您的程序利用多路复用 I/O（multiplexed I/O）。选择和多路复用将在第四章中予以讨论。关于怎样将 sockets 置于非阻塞模式的细节会在 3.5 节中涉及。

3.1.3 关闭通道

与缓冲区不同，通道不能被重复使用。一个打开的通道即代表与一个特定 I/O 服务的特定连接并封装该连接的状态。当通道关闭时，那个连接会丢失，然后通道将不再连接任何东西。

```
package java.nio.channels;
public interface Channel
{
    public boolean isOpen( );
    public void close( ) throws IOException;
}
```

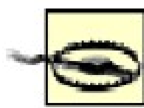
调用通道的*close( )*方法时，可能会导致在通道关闭底层I/O服务的过程中线程暂时阻塞[7]，哪怕该通道处于非阻塞模式。通道关闭时的阻塞行为（如果有的话）是高度取决于操作系统或者文件系统的。在一个通道上多次调用*close( )*方法是没有坏处的，但是如果第一个线程在*close( )*方法中阻塞，那么在它完成关闭通道之前，任何其他调用*close( )*方法都会阻塞。后续在该已关闭的通道上调用*close( )*不会产生任何操作，只会立即返回。

可以通过 *isOpen( )*方法来测试通道的开放状态。如果返回 `true` 值，那么该通道可以使用。如果返回 `false` 值，那么该通道已关闭，不能再被使用。尝试进行任何需要通道处于开放状态作为前提的操作，如读、写等都会导致 *ClosedChannelException* 异常。

通道引入了一些与关闭和中断有关的新行为。如果一个通道实现 *InterruptibleChannel* 接口（参见图 3-2），它的行为以下述语义为准：如果一个线程在一个通道上被阻塞并且同时被中断（由调用该被阻塞线程的 *interrupt( )*方法的另一个线程中断），那么该通道将被关闭，该被阻塞线程也会产生一个 *ClosedByInterruptException* 异常。

此外，假如一个线程的 interrupt status 被设置并且该线程试图访问一个通道，那么这个通道将立即被关闭，同时将抛出相同的 *ClosedByInterruptException* 异常。线程的 interrupt status 在线程的 *interrupt( )*方法被调用时会被设置。我们可以使用 *isInterrupted( )*来测试某个线程当前的 interrupt status。当前线程的 interrupt status 可以通过调用静态的 *Thread.interrupted( )*方法清除。

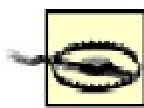
[7] Socket 通道关闭会花费较长时间，具体时耗取决于操作系统的网络实现。在输出内容被提取时，一些网络协议堆栈可能会阻塞通道的关闭。



请不要将在 *Channels* 上休眠的中断线程同在 *Selectors* 上休眠的中断线程混淆。前者会关闭通道，而后者则不会。不过，如果您的线程在 *Selector* 上休眠时被中断，那它的 interrupt status 会被设置。假设那个线程接着又访问一个 *Channel*，则该通道会被关闭。关于选择器（*Selectors*）我们会在第四章中讨论。

仅仅因为休眠在其上的线程被中断就关闭通道，这看起来似乎过于苛刻了。不过这却是 NIO 架构师们所做出的明确的设计决定。经验表明，想要在所有的操作系统上一致而可靠地处理被中断的 I/O 操作是不可能的。“在全部平台上提供确定的通道行为”这一需求导致了“当 I/O 操作被中断时总是关闭通道”这一设计选择。这个选择被认为是可接受的，因为大部分时候一个线程被中断就是希望以此来关闭通道。`java.nio` 包中强制使用此行为来避免因操作系统独特性而导致的困境，因为该困境对 I/O 区域而言是极其危险的。这也是为增强健壮性（robustness）而采用的一种经典的权衡。

可中断的通道也是可以异步关闭的。实现 *InterruptibleChannel* 接口的通道可以在任何时候被关闭，即使有另一个被阻塞的线程在等待该通道上的一个 I/O 操作完成。当一个通道被关闭时，休眠在该通道上的所有线程都将被唤醒并接收到一个 *AsynchronousCloseException* 异常。接着通道就被关闭并将不再可用。



一开始的 NIO 版本 JDK1.4.0 有几个与中断的通道操作以及异步可关闭性相关的严重缺陷。这些问题期待在 1.4.1 版本中的到解决。在 1.4.0 版本中，当休眠在一个通道 I/O 操作上的线程被阻塞或者该通道被另一个线程关闭时，这些线程也许不能确保都能被唤醒。因此在使用依靠该行为的操作时请务必小心。

不实现 *InterruptibleChannel* 接口的通道一般都是不进行底层本地代码实现的有特殊用途的通道。这些也许是永远不会阻塞的特殊用途通道，如旧系统数据流的封装包或不能实现可中断语义的 writer 类等。（参见 3.7 节）

3.2 Scatter/Gather

通道提供了一种被称为 Scatter/Gather 的重要新功能（有时也被称为矢量 I/O）。Scatter/Gather 是一个简单却强大的概念（参见 1.4.1.1 节），它是指在多个缓冲区上实现一个简单的 I/O 操作。对于一个 write 操作而言，数据是从几个缓冲区按顺序抽取（称为 gather）并沿着通道发送的。缓冲区本身并不需要具备这种 gather 的能力（通常它们也没有此能力）。该 gather 过程的效果就好比全部缓冲区的内容被连结起来，并在发送数据前存放到一个大的缓冲区中。对于 read 操作而言，从通道读取的数据会按顺序被散布（称为 scatter）到多个缓冲区，将每个缓冲区填满直至通道中的数

据或者缓冲区的最大空间被消耗完。

大多数现代操作系统都支持本地矢量 I/O（native vectored I/O）。当您在一个通道上请求一个 Scatter/Gather 操作时，该请求会被翻译为适当的本地调用来直接填充或抽取缓冲区。这是一个很大的进步，因为减少或避免了缓冲区拷贝和系统调用。Scatter/Gather 应该使用直接的 *ByteBuffers* 以从本地 I/O 获取最大性能优势。

将 scatter/gather 接口添加到图 3-3 的 UML 类图中可以得到图 3-4。下面的代码描述了 scatter 是如何扩展读操作的，以及 gather 是如何基于写操作构建的：

```
public interface ScatteringByteChannel
    extends ReadableByteChannel
{
    public long read (ByteBuffer [] dsts)
        throws IOException;
    public long read (ByteBuffer [] dsts, int offset, int length)
        throws IOException;
}
public interface GatheringByteChannel
    extends WritableByteChannel
{
    public long write(ByteBuffer[] srcs)
        throws IOException;
    public long write(ByteBuffer[] srcs, int offset, int length)
        throws IOException;
}
```

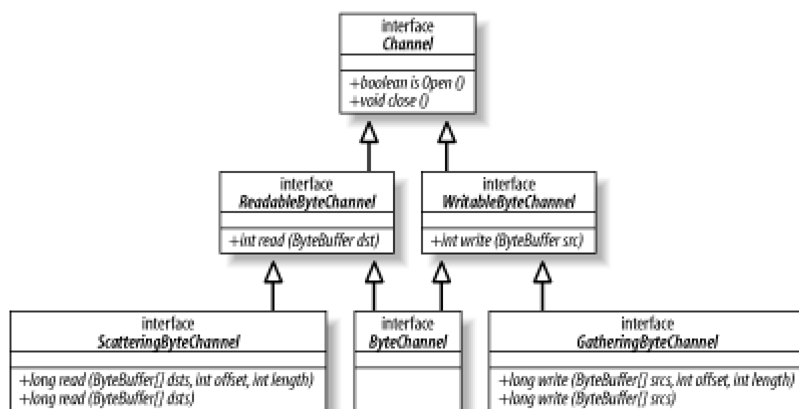


图 3-4 Scatter/Gather 接口

从上图您可以看到，这两个接口都添加了两种以缓冲区阵列作为参数的新方法。另外，每种方法都提供了一种带 offset 和 length 参数的形式。让我们先来理解一下怎样使用方法的简单形式。在下面的代码中，我们假定 channel 连接到一个有 48 字节数据等待读取的 socket 上：

```
ByteBuffer header = ByteBuffer.allocateDirect (10);
ByteBuffer body = ByteBuffer.allocateDirect (80);
ByteBuffer [] buffers = { header, body };
int bytesRead = channel.read (buffers);
```

一旦 *read( )*方法返回，bytesRead 就被赋予值 48，header 缓冲区将包含前 10 个从通道读取的字节而 body 缓冲区则包含接下来的 38 个字节。通道会自动地将数据 scatter 到这两个缓冲区中。缓冲区已经被填充了（尽管此例中 body 缓冲区还有空间填充更多数据），那么将需要被 flip 以便其中数据可以被抽取。在类似这样的例子中，我们可能并不会费劲去 flip 这个 header 缓冲区而是以绝对 get 的方式随机访问它以检查各种 header 字段；不过 body 缓冲区会被 flip 并传递到另一个通道的 *write( )*方法上，然后在通道上发送出去。例如：

```
switch (header.getShort(0)) {
    case TYPE_PING:
        break;
    case TYPE_FILE:
        body.flip( );
        fileChannel.write (body);
        break;
    default:
        logUnknownPacket (header.getShort(0), header.getLong(2), body);
        break;
}
```

同样，很简单地，我们可以用一个 gather 操作将多个缓冲区的数据组合并发送出去。使用相同的缓冲区，我们可以像下面这样汇总数据并在一个 socket 通道上发送包：

```
body.clear( );
body.put("FOO".getBytes()).flip( ); // "FOO" as bytes
header.clear( );
header.putShort (TYPE_FILE).putLong (body.limit()).flip( );
long bytesWritten = channel.write (buffers);
```

以上代码从传递给 *write( )*方法的 buffers 阵列所引用的缓冲区中 gather 数据，然后沿着通道

发送了总共 13 个字节。

图 3-5 描述了一个 gather 写操作。数据从缓冲区阵列引用的每个缓冲区中 gather 并被组合成沿着通道发送的字节流。

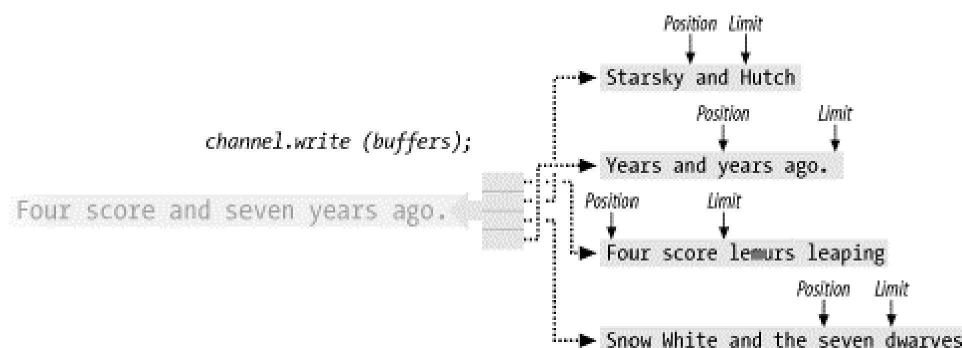


图 3-5 一个使用了四个缓冲区的 gather 写操作

图 3-6 描述了一个 scatter 读操作。从通道传输来的数据被 scatter 到所列缓冲区，依次填充每个缓冲区（从缓冲区的 position 处开始到 limit 处结束）。这里显示的 position 和 limit 值是读操作开始之前的。

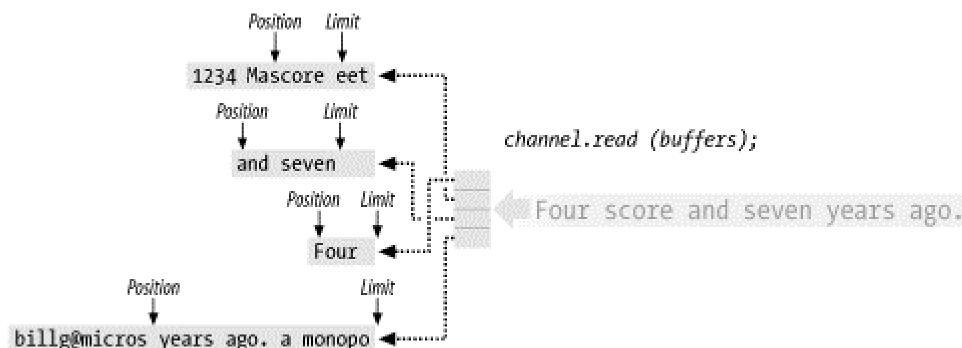


图 3-6 一个使用了四个缓冲区的 scatter 读操作

带 `offset` 和 `length` 参数版本的 *read( )* 和 *write( )*方法使得我们可以使用缓冲区阵列的子集缓冲区。这里的 `offset` 值指哪个缓冲区将开始被使用，而不是指数据的 offset。这里的 length 参数指示要使用的缓冲区数量。举个例子，假设我们有一个五元素的 `fiveBuffers` 阵列，它已经被初始化并引用了五个缓冲区，下面的代码将会写第二个、第三个和第四个缓冲区的内容：

```
int bytesRead = channel.write (fiveBuffers, 1, 3);
```

使用得当的话，Scatter/Gather 会是一个极其强大的工具。它允许您委托操作系统来完成辛苦活：将读取到的数据分开存放到多个存储桶（bucket）或者将不同的数据区块合并成一个整体。这是一个巨大的成就，因为操作系统已经被高度优化来完成此类工作了。它节省了您来回移动数据的

工作，也就避免了缓冲区拷贝和减少了您需要编写、调试的代码数量。既然您基本上通过提供数据容器引用来组合数据，那么按照不同的组合构建多个缓冲区阵列引用，各种数据区块就可以以不同的方式来组合了。例 3-2 很好地诠释了这一点：

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.channels.GatheringByteChannel;
import java.io.FileOutputStream;
import java.util.Random;
import java.util.List;
import java.util.LinkedList;
/**
  * Demonstrate gathering write using many buffers.
  *
  * @author Ron Hitchens (ron@ronsoft.com)
  */
public class Marketing
{
    private static final String DEMOGRAPHIC = "blahblah.txt";
    // "Leverage frictionless methodologies"
    public static void main (String [] argv)
        throws Exception
    {
        int reps = 10;
        if (argv.length > 0) {
            reps = Integer.parseInt (argv [0]);
        }
        FileOutputStream fos = new FileOutputStream (DEMOGRAPHIC);
        GatheringByteChannel gatherChannel = fos.getChannel( );
        // Generate some brilliant marcom, er, repurposed content
        ByteBuffer [] bs = utterBS (reps);
        // Deliver the message to the waiting market
        while (gatherChannel.write (bs) > 0) {
            // Empty body
            // Loop until write( ) returns zero
        }
        System.out.println ("Mindshare paradigms synergized to "
            + DEMOGRAPHIC);
```

```
    fos.close( );
}
// ------------------------------------------------
// These are just representative; add your own
private static String [] col1 = {
"Aggregate", "Enable", "Leverage",
"Facilitate", "Synergize", "Repurpose",
"Strategize", "Reinvent", "Harness"
};
private static String [] col2 = {
    "cross-platform", "best-of-breed", "frictionless",
    "ubiquitous", "extensible", "compelling",
    "mission-critical", "collaborative", "integrated"
};
private static String [] col3 = {
    "methodologies", "infomediaries", "platforms",
    "schemas", "mindshare", "paradigms",
    "functionalities", "web services", "infrastructures"
};
private static String newline = System.getProperty ("line.separator");
// The Marcom-atic 9000
private static ByteBuffer [] utterBS (int howMany)
    throws Exception
{
    List list = new LinkedList( );
    for (int i = 0; i < howMany; i++) {
        list.add (pickRandom (col1, " "));
        list.add (pickRandom (col2, " "));
        list.add (pickRandom (col3, newline));
    }
    ByteBuffer [] bufs = new ByteBuffer [list.size( )];
    list.toArray (bufs);
    return (bufs);
}
// The communications director
private static Random rand = new Random( );
// Pick one, make a buffer to hold it and the suffix, load it with
// the byte equivalent of the strings (will not work properly for
```

```
    // non-Latin characters), then flip the loaded buffer so it's ready
    // to be drained
    private static ByteBuffer pickRandom (String [] strings, String suffix)
        throws Exception
    {
        String string = strings [rand.nextInt (strings.length)];
        int total = string.length() + suffix.length( );
        ByteBuffer buf = ByteBuffer.allocate (total);
        buf.put (string.getBytes ("US-ASCII"));
        buf.put (suffix.getBytes ("US-ASCII"));
        buf.flip( );
        return (buf);
    }
}
```

例 3-2 以 gather 写操作来集合多个缓冲区的数据

下面是实现 `Marketing` 类的输出。虽然这种输出没什么意义，但是 gather 写操作却能让我们非常高效地把它生成出来。

```
Aggregate compelling methodologies
Harness collaborative platforms
Aggregate integrated schemas
Aggregate frictionless platforms
Enable integrated platforms
Leverage cross-platform functionalities
Harness extensible paradigms
Synergize compelling infomediaries
Repurpose cross-platform mindshare
Facilitate cross-platform infomediaries
```

3.3 文件通道

直到现在，我们都还只是在泛泛地讨论通道，比如讨论那些对所有通道都适用的内容。是时候具体点了，本节我们来讨论文件通道（socket 通道将在下一节讨论）。从图 3-7 可以发现，*FileChannel* 类可以实现常用的 read，write 以及 scatter/gather 操作，同时它也提供了很多专用于文件的新方法。这些方法中的许多都是我们所熟悉的文件操作，不过其他的您可能之前并未接触过。现在我们将在此对它们全部予以讨论。

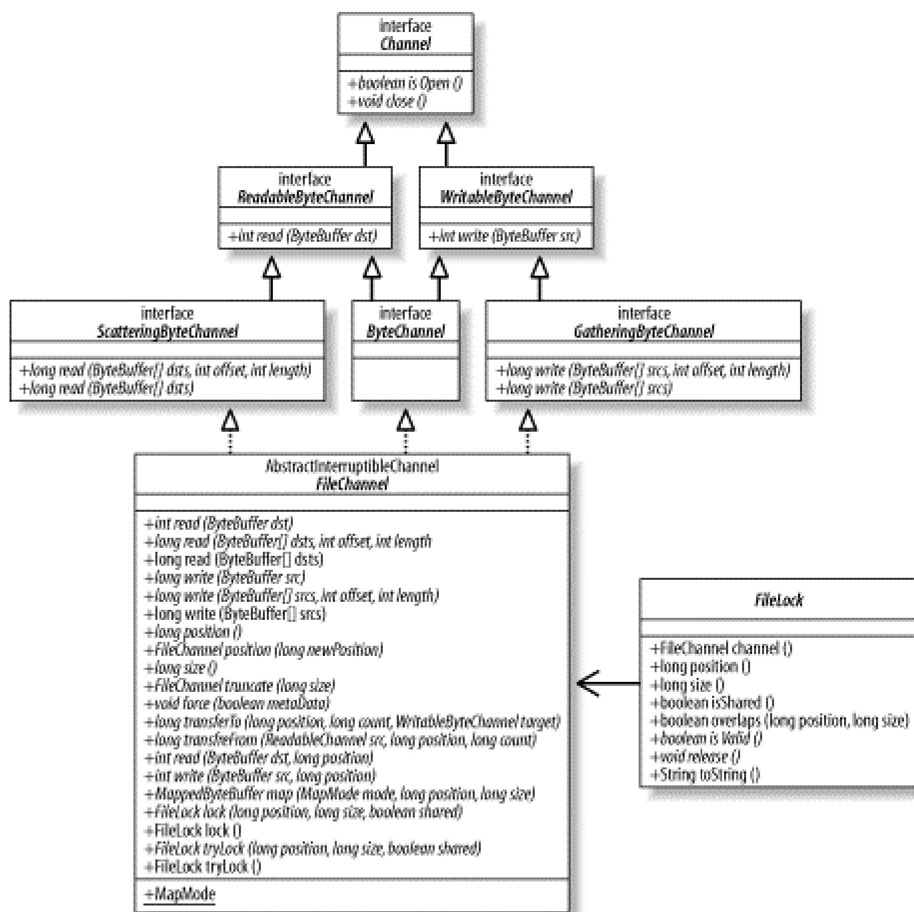


图 3-7 FileChannel 类层次结构

文件通道总是阻塞式的，因此不能被置于非阻塞模式。现代操作系统都有复杂的缓存和预取机制，使得本地磁盘 I/O 操作延迟很少。网络文件系统一般而言延迟会多些，不过却也因该优化而受益。面向流的 I/O 的非阻塞范例对于面向文件的操作并无多大意义，这是由文件 I/O 本质上的不同性质造成的。对于文件 I/O，最强大之处在于异步 I/O（asynchronous I/O），它允许一个进程可以从操作系统请求一个或多个 I/O 操作而不必等待这些操作的完成。发起请求的进程之后会收到它请求的 I/O 操作已完成的通知。异步 I/O 是一种高级性能，当前的很多操作系统都还不具备。以后的 NIO 增强也会把异步 I/O 纳入考虑范围。

我们在 3.1.1 节中提到，*FileChannel*对象不能直接创建。一个*FileChannel*实例只能通过在一个打开的file对象（*RandomAccessFile*、*FileInputStream*或 *FileOutputStream*）上调用*getChannel( )*方法获取 [8]。调用*getChannel( )*方法会返回一个连接到相同文件的*FileChannel*对象且该*FileChannel*对象具有与file对象相同的访问权限，然后您就可以使用该通道对象来利用强大的*FileChannel* API了：

---

[8] JSR 51 也定义了对于一个扩展的文件系统接口 API 的需求。JDK 1.4 版本中尚未实现该 API，不过 JDK 1.5 版本中有望实现。一旦此改进的文件系统 API 就位，它将很可能成为 *FileChannel* 对象首选的来源。

```
package java.nio.channels;
public abstract class FileChannel
    extends AbstractChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
    // All methods listed here can throw java.io.IOException
    public abstract int read (ByteBuffer dst, long position)
    public abstract int write (ByteBuffer src, long position)
    public abstract long size( )
    public abstract long position( )
    public abstract void position (long newPosition)
    public abstract void truncate (long size)
    public abstract void force (boolean metaData)
    public final FileLock lock( )
    public abstract FileLock lock (long position, long size,
        boolean shared)
    public final FileLock tryLock( )
    public abstract FileLock tryLock (long position, long size,
        boolean shared)
    public abstract MappedByteBuffer map (MapMode mode, long position,
        long size)
    public static class MapMode
    {
        public static final MapMode READ_ONLY
        public static final MapMode READ_WRITE
        public static final MapMode PRIVATE
    }
    public abstract long transferTo (long position, long count,
            WritableByteChannel target)
    public abstract long transferFrom (ReadableByteChannel src,
            long position, long count)
}
```

上面的代码中给出了 *FileChannel* 类引入的新 API 方法。所有这些方法都可以抛出 *java.io.IOException* 异常，不过抛出语句并未在此列出。

同大多数通道一样，只要有可能，*FileChannel* 都会尝试使用本地 I/O 服务。*FileChannel* 类本

身是抽象的，您从 *getChannel( )*方法获取的实际对象是一个具体子类（subclass）的一个实例（instance），该子类可能使用本地代码来实现以上 API 方法中的一些或全部。

FileChannel 对象是线程安全（thread-safe）的。多个进程可以在同一个实例上并发调用方法而不会引起任何问题，不过并非所有的操作都是多线程的（multithreaded）。影响通道位置或者影响文件大小的操作都是单线程的（single-threaded）。如果有一个线程已经在执行会影响通道位置或文件大小的操作，那么其他尝试进行此类操作之一的线程必须等待。并发行为也会受到底层的操作系统或文件系统影响。

同大多数 I/O 相关的类一样，*FileChannel* 是一个反映 Java 虚拟机外部一个具体对象的抽象。*FileChannel* 类保证同一个 Java 虚拟机上的所有实例看到的某个文件的视图均是一致的，但是 Java 虚拟机却不能对超出它控制范围的因素提供担保。通过一个 *FileChannel* 实例看到的某个文件的视图同通过一个外部的非 Java 进程看到的该文件的视图可能一致，也可能不一致。多个进程发起的并发文件访问的语义高度取决于底层的操作系统和（或）文件系统。一般而言，由运行在不同 Java 虚拟机上的 *FileChannel* 对象发起的对某个文件的并发访问和由非 Java 进程发起的对该文件的并发访问是一致的。

3.3.1 访问文件

每个 *FileChannel* 对象都同一个文件描述符（file descriptor）有一对一的关系，所以上面列出的 API 方法与在您最喜欢的 POSIX（可移植操作系统接口）兼容的操作系统上的常用文件 I/O 系统调用紧密对应也就不足为怪了。名称也许不尽相同，不过常见的 suspect（“可疑分子”）都被集中起来了。您可能也注意到了上面列出的 API 方法同 `java.io` 包中 *RandomAccessFile* 类的方法的相似之处了。本质上讲，*RandomAccessFile* 类提供的是同样的抽象内容。在通道出现之前，底层的文件操作都是通过 *RandomAccessFile* 类的方法来实现的。*FileChannel* 模拟同样的 I/O 服务，因此它的 API 自然也是很相似的。

为了便于比较，表 3-1 列出了 *FileChannel*、*RandomAccessFile* 和 POSIX I/O system calls 三者在方法上的对应关系。

表 3-1 File I/O API 比较

FileChannel	RandomAccessFile	POSIX system call
read( )	*read( )*	*read( )*
write( )	*write( )*	*write( )*
size( )	*length( )*	*fstat( )*
position( )	*getFilePointer( )*	*lseek( )*
position (long newPosition)	*seek( )*	*lseek( )*
truncate( )	*setLength( )*	*ftruncate( )*

force( )	*getFD().sync( )*	*fsync( )*

让我们来进一步看下基本的文件访问方法（请记住这些方法都可以抛出 *java.io.IOException* 异常）：

```
public abstract class FileChannel
    extends AbstractChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
    public abstract long position( )
    public abstract void position (long newPosition)
    public abstract int read (ByteBuffer dst)
    public abstract int read (ByteBuffer dst, long position)
    public abstract int write (ByteBuffer src)
    public abstract int write (ByteBuffer src, long position)
    public abstract long size( )
    public abstract void truncate (long size)
    public abstract void force (boolean metaData)
}
```

同底层的文件描述符一样，每个 *FileChannel* 都有一个叫“file position”的概念。这个 position 值决定文件中哪一处的数据接下来将被读或者写。从这个方面看，*FileChannel* 类同缓冲区很类似，并且 *MappedByteBuffer* 类使得我们可以通过 *ByteBuffer* API 来访问文件数据（我们会在后面的章节中了解到这一点）。

您可以从前面的API清单中看到，有两种形式的*position( )*方法。第一种，不带参数的，返回当前文件的position值。返回值是一个长整型（long），表示文件中的当前字节位置 [9]。

第二种形式的 *position( )*方法带一个 `long`（长整型）参数并将通道的 position 设置为指定值。如果尝试将通道 position 设置为一个负值会导致 *java.lang.IllegalArgumentException* 异常，不过可以把 position 设置到超出文件尾，这样做会把 position 设置为指定值而不改变文件大小。假如在将 position 设置为超出当前文件大小时实现了一个 *read( )*方法，那么会返回一个文件尾（end-of-file）条件；倘若此时实现的是一个 *write( )*方法则会引起文件增长以容纳写入的字节，具体行为类似于实现一个绝对 *write( )*并可能导致出现一个文件空洞(file hole，参见“文件空洞究竟是什么？”)。

---

[9] 一个有符号长整型（signed `long`）值能代表多达 9,223,372,036,854,775,807 字节的文件大小。这个数据量差不多是 840 万 TB，可以在您本地电脑上填满大约 9000 万个 100-GB 的磁盘驱动器。

文件空洞究竟是什么？

当磁盘上一个文件的分配空间小于它的文件大小时会出现“文件空洞”。对于内容稀疏的文件，大多数现代文件系统只为实际写入的数据分配磁盘空间（更准确地说，只为那些写入数据的文件系统页分配空间）。假如数据被写入到文件中非连续的位置上，这将导致文件出现在逻辑上不包含数据的区域（即“空洞”）。例如，下面的代码可能产生一个如图 3-8 所示的文件：

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.io.File;
import java.io.RandomAccessFile;
import java.io.IOException;
/**
  * Create a file with holes in it
  *
  * @author Ron Hitchens (ron@ronsoft.com)
  */
public class FileHole
{
        public static void main (String [] argv)
        throws IOException
        {
                // Create a temp file, open for writing, and get
                // a FileChannel
                File temp = File.createTempFile ("holy", null);
                RandomAccessFile file = new RandomAccessFile (temp, "rw");
                FileChannel channel = file.getChannel( );
                // Create a working buffer
                ByteBuffer byteBuffer = ByteBuffer.allocateDirect (100);
                putData (0, byteBuffer, channel);
                putData (5000000, byteBuffer, channel);
                putData (50000, byteBuffer, channel);
                // Size will report the largest position written, but
                // there are two holes in this file. This file will
                // not consume 5 MB on disk (unless the filesystem is
```

```
                    // extremely brain-damaged)
                    System.out.println ("Wrote temp file '" + temp.getPath( )
                              + "', size=" + channel.size( ));
                    channel.close( );
                    file.close( );
          }
          private static void putData (int position, ByteBuffer buffer,
                    FileChannel channel)
                    throws IOException
          {
                    String string = "*<-- location " + position;
                    buffer.clear( );
                    buffer.put (string.getBytes ("US-ASCII"));
                    buffer.flip( );
                    channel.position (position);
                    channel.write (buffer);
          }
}
```

如果该文件被顺序读取的话，所有空洞都会被“0”填充但不占用磁盘空间。读取该文件的进程会看到 5,000,021 个字节，大部分字节都以“0”表示。试试在该文件上运行 *strings* 命令，看看您会得到什么。再试试将文件大小的值提高到 50 或 100MB，看看您的全部磁盘空间消耗以及顺序扫描该文件所需时间会发生何种变化（前者不会改变，但是后者将有非常大的增加）。

FileChannel 位置（position）是从底层的文件描述符获得的，该 position 同时被作为通道引用获取来源的文件对象共享。这也就意味着一个对象对该 position 的更新可以被另一个对象看到：

```
RandomAccessFile randomAccessFile = new RandomAccessFile ("filename", "r");
// Set the file position
randomAccessFile.seek (1000);
// Create a channel from the file
FileChannel fileChannel = randomAccessFile.getChannel( );
// This will print "1000"
System.out.println ("file pos: " + fileChannel.position( ));
// Change the position using the RandomAccessFile object
randomAccessFile.seek (500);
```

```
// This will print "500"
System.out.println ("file pos: " + fileChannel.position( ));
// Change the position using the FileChannel object
fileChannel.position (200);
// This will print "200"
System.out.println ("file pos: " + randomAccessFile.getFilePointer( ));
```

类似于缓冲区的 *get( )* 和 *put( )*方法，当字节被 *read( )*或 *write( )*方法传输时，文件 position 会自动更新。如果 position 值达到了文件大小的值（文件大小的值可以通过 *size( )*方法返回），*read( )*方法会返回一个文件尾条件值（-1）。可是，不同于缓冲区的是，如果实现 *write( )*方法时 position 前进到超过文件大小的值，该文件会扩展以容纳新写入的字节。

同样类似于缓冲区，也有带 position 参数的绝对形式的 *read( )*和 *write( )*方法。这种绝对形式的方法在返回值时不会改变当前的文件 position。由于通道的状态无需更新，因此绝对的读和写可能会更加有效率，操作请求可以直接传到本地代码。更妙的是，多个线程可以并发访问同一个文件而不会相互产生干扰。这是因为每次调用都是原子性的（atomic），并不依靠调用之间系统所记住的状态。

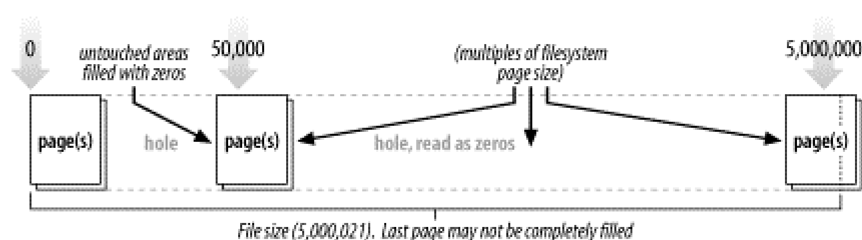


图 3-8 有两个空洞的磁盘文件

尝试在文件末尾之外的 position 进行一个绝对读操作，*size( )*方法会返回一个 end-of-file。在超出文件大小的 position 上做一个绝对 *write( )*会导致文件增加以容纳正在被写入的新字节。文件中位于之前 end-of-file 位置和新添加的字节起始位置之间区域的字节的值不是由 *FileChannel* 类指定，而是在大多数情况下反映底层文件系统的语义。取决于操作系统和（或）文件系统类型，这可能会导致在文件中出现一个空洞。

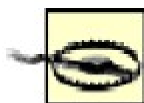
当需要减少一个文件的 size 时，*truncate( )*方法会砍掉您所指定的新 size 值之外的所有数据。如果当前 size 大于新 size，超出新 size 的所有字节都会被悄悄地丢弃。如果提供的新 size 值大于或等于当前的文件 size 值，该文件不会被修改。这两种情况下，*truncate( )*都会产生副作用：文件的 position 会被设置为所提供的新 size 值。

```
public abstract class FileChannel
    extends AbstractChannel
```

```
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
    public abstract void truncate (long size)
    public abstract void force (boolean metaData)
}
```

上面列出的最后一个 API 是 *force( )*。该方法告诉通道强制将全部待定的修改都应用到磁盘的文件上。所有的现代文件系统都会缓存数据和延迟磁盘文件更新以提高性能。调用 *force( )*方法要求文件的所有待定修改立即同步到磁盘。

如果文件位于一个本地文件系统，那么一旦 *force( )*方法返回，即可保证从通道被创建（或上次调用 *force( )*）时起的对文件所做的全部修改已经被写入到磁盘。对于关键操作如事务（transaction）处理来说，这一点是非常重要的，可以保证数据完整性和可靠的恢复。然而，如果文件位于一个远程的文件系统，如 NFS 上，那么不能保证待定修改一定能同步到永久存储器（permanent storage）上，因 Java 虚拟机不能做操作系统或文件系统不能实现的承诺。如果您的程序在面临系统崩溃时必须维持数据完整性，先去验证一下您在使用的操作系统和（或）文件系统在同步修改方面是可以依赖的。



对于需要特别保证数据完整性的应用程序，请务必验证一下您计划部署该程序的操作环境的性能。

*force( )*方法的布尔型参数表示在方法返回值前文件的元数据（metadata）是否也要被同步更新到磁盘。元数据指文件所有者、访问权限、最后一次修改时间等信息。大多数情形下，该信息对数据恢复而言是不重要的。给 *force( )*方法传递 `false` 值表示在方法返回前只需要同步文件数据的更改。大多数情形下，同步元数据要求操作系统进行至少一次额外的底层 I/O 操作。一些大数量事务处理程序可能通过在每次调用 *force( )*方法时不要求元数据更新来获取较高的性能提升，同时也不会牺牲数据完整性。

3.3.2 文件锁定

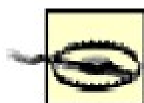
在 JDK 1.4 版本之前，Java I/O 模型都未能提供文件锁定（file locking），缺少这一特性让人们很头疼。绝大多数现代操作系统早就有了文件锁定功能，而直到 JDK 1.4 版本发布时 Java 编程人员才可以使用文件锁（file lock）。在集成许多其他非 Java 程序时，文件锁定显得尤其重要。此外，它在判优（判断多个访问请求的优先级别）一个大系统的多个 Java 组件发起的访问时也很有价值。

我们在第一章中讨论到，锁（lock）可以是共享的（shared）或独占的（exclusive）。本节中描述的文件锁定特性在很大程度上依赖本地的操作系统实现。并非所有的操作系统和文件系统都支持共享文件锁。对于那些不支持的，对一个共享锁的请求会被自动提升为对独占锁的请求。这可以保证准确性却可能严重影响性能。举个例子，仅使用独占锁将会串行化图 1-7 中所列的全部 reader 进程。如果您计划部署程序，请确保您了解所用操作系统和文件系统的文件锁定行为，因为这将严重影响您的设计选择。

另外，并非所有平台都以同一个方式来实现基本的文件锁定。在不同的操作系统上，甚至在同一个操作系统的不同文件系统上，文件锁定的语义都会有所差异。一些操作系统仅提供劝告锁定（advisory locking），一些仅提供独占锁（exclusive locks），而有些操作系统可能两种锁都提供。您应该总是按照劝告锁的假定来管理文件锁，因为这是最安全的。但是如能了解底层操作系统如何执行锁定也是非常好的。例如，如果所有的锁都是强制性的（mandatory）而您不及时释放您获得的锁的话，运行在同一操作系统上的其他程序可能会受到影响。

有关 *FileChannel* 实现的文件锁定模型的一个重要注意项是：锁的对象是文件而不是通道或线程，这意味着文件锁不适用于判优同一台 Java 虚拟机上的多个线程发起的访问。

如果一个线程在某个文件上获得了一个独占锁，然后第二个线程利用一个单独打开的通道来请求该文件的独占锁，那么第二个线程的请求会被批准。但如果这两个线程运行在不同的 Java 虚拟机上，那么第二个线程会阻塞，因为锁最终是由操作系统或文件系统来判优的并且几乎总是在进程级而非线程级上判优。锁都是与一个文件关联的，而不是与单个的文件句柄或通道关联。



锁与文件关联，而不是与通道关联。我们使用锁来判优外部进程，而不是判优同一个 Java 虚拟机上的线程。

文件锁旨在在进程级别上判优文件访问，比如在主要的程序组件之间或者在集成其他供应商的组件时。如果您需要控制多个 Java 线程的并发访问，您可能需要实施您自己的、轻量级的锁定方案。那种情形下，内存映射文件（本章后面会进行详述）可能是一个合适的选择。

现在让我们来看下与文件锁定有关的 *FileChannel* API 方法：

```
public abstract class FileChannel
    extends AbstractChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
```

```
    public final FileLock lock( )
    public abstract FileLock lock (long position, long size,
        boolean shared)
    public final FileLock tryLock( )
    public abstract FileLock tryLock (long position, long size,
        boolean shared)
}
```

这次我们先看带参数形式的 *lock( )*方法。锁是在文件内部区域上获得的。调用带参数的 *Lock( )* 方法会指定文件内部锁定区域的开始 position 以及锁定区域的 size。第三个参数 shared 表示您想获取的锁是共享的（参数值为 true）还是独占的（参数值为 false）。要获得一个共享锁，您必须先以只读权限打开文件，而请求独占锁时则需要写权限。另外，您提供的 position 和 size 参数的值不能是负数。

锁定区域的范围不一定要限制在文件的 size 值以内，锁可以扩展从而超出文件尾。因此，我们可以提前把待写入数据的区域锁定，我们也可以锁定一个不包含任何文件内容的区域，比如文件最后一个字节以外的区域。如果之后文件增长到达那块区域，那么您的文件锁就可以保护该区域的文件内容了。相反地，如果您锁定了文件的某一块区域，然后文件增长超出了那块区域，那么新增加的文件内容将不会受到您的文件锁的保护。

不带参数的简单形式的 *lock( )*方法是一种在整个文件上请求独占锁的便捷方法，锁定区域等于它能达到的最大范围。该方法等价于：

```
fileChannel.lock (0L, Long.MAX_VALUE, false);
```

如果您正请求的锁定范围是有效的，那么 *lock( )*方法会阻塞，它必须等待前面的锁被释放。假如您的线程在此情形下被暂停，该线程的行为受中断语义（类似我们在 3.1.3 节中所讨论的）控制。如果通道被另外一个线程关闭，该暂停线程将恢复并产生一个 *AsynchronousCloseException* 异常。假如该暂停线程被直接中断（通过调用它的 *interrupt( )*方法），它将醒来并产生一个 *FileLockInterruptionException* 异常。如果在调用 *lock( )*方法时线程的 interrupt status 已经被设置，也会产生 *FileLockInterruptionException* 异常。

在上面的 API 列表中有两个名为 *tryLock( )*的方法，它们是 *lock( )*方法的非阻塞变体。这两个 *tryLock( )*和 *lock( )*方法起相同的作用，不过如果请求的锁不能立即获取到则会返回一个 null。

您可以看到，*lock( )*和 *tryLock( )*方法均返回一个 *FileLock* 对象。以下是完整的 *FileLock* API：

```
public abstract class FileLock
{
```

```
    public final FileChannel channel( )
    public final long position( )
    public final long size( )
    public final boolean isShared( )
    public final boolean overlaps (long position, long size)
    public abstract boolean isValid( );
    public abstract void release( ) throws IOException;
}
```

FileLock 类封装一个锁定的文件区域。*FileLock* 对象由 *FileChannel* 创建并且总是关联到那个特定的通道实例。您可以通过调用 *channel( )*方法来查询一个 lock 对象以判断它是由哪个通道创建的。

一个 *FileLock* 对象创建之后即有效，直到它的 *release( )*方法被调用或它所关联的通道被关闭或 Java 虚拟机关闭时才会失效。我们可以通过调用 *isValid( )*布尔方法来测试一个锁的有效性。一个锁的有效性可能会随着时间而改变，不过它的其他属性——位置（position）、范围大小（size）和独占性（exclusivity）——在创建时即被确定，不会随着时间而改变。

您可以通过调用 *isShared( )*方法来测试一个锁以判断它是共享的还是独占的。如果底层的操作系统或文件系统不支持共享锁，那么该方法将总是返回 `false` 值，即使您申请锁时传递的参数值是 `true`。假如您的程序依赖共享锁定行为，请测试返回的锁以确保您得到了您申请的锁类型。*FileLock* 对象是线程安全的，多个线程可以并发访问一个锁对象。

最后，您可以通过调用 *overlaps( )*方法来查询一个 *FileLock* 对象是否与一个指定的文件区域重叠。这将使您可以迅速判断您拥有的锁是否与一个感兴趣的区域（region of interest）有交叉。不过即使返回值是 `false` 也不能保证您就一定能在期望的区域上获得一个锁，因为 Java 虚拟机上的其他地方或者外部进程可能已经在该期望区域上有一个或多个锁了。您最好使用 *tryLock( )*方法确认一下。

尽管一个 *FileLock* 对象是与某个特定的 *FileChannel* 实例关联的，它所代表的锁却是与一个底层文件关联的，而不是与通道关联。因此，如果您在使用完一个锁后而不释放它的话，可能会导致冲突或者死锁。请小心管理文件锁以避免出现此问题。一旦您成功地获取了一个文件锁，如果随后在通道上出现错误的话，请务必释放这个锁。推荐使用类似下面的代码形式：

```
FileLock lock = fileChannel.lock( )
try {
        <perform read/write/whatever on channel>
} catch (IOException) [
        <handle unexpected exception>
```

```
} finally {
        lock.release( )
}
```

例 3-3 中的代码使用共享锁实现了 reader 进程，使用独占锁实现了 writer 进程，图 1-7 和图 1-8 对此有诠释。由于锁是与进程而不是 Java 线程关联的，您将需要运行该程序的多个拷贝。先从一个 writer 和两个或更多的 readers 开始，我们来看下不同类型的锁是如何交互的。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.io.RandomAccessFile;
import java.util.Random;
/**
 * Test locking with FileChannel.
 * Run one copy of this code with arguments "-w /tmp/locktest.dat"
 * and one or more copies with "-r /tmp/locktest.dat" to see the
 * interactions of exclusive and shared locks. Note how too many
 * readers can starve out the writer.
 * Note: The filename you provide will be overwritten. Substitute
 * an appropriate temp filename for your favorite OS.
 *
 * Created April, 2002
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class LockTest
{
    private static final int SIZEOF_INT = 4;
    private static final int INDEX_START = 0;
    private static final int INDEX_COUNT = 10;
    private static final int INDEX_SIZE = INDEX_COUNT * SIZEOF_INT;
    private ByteBuffer buffer = ByteBuffer.allocate (INDEX_SIZE);
    private IntBuffer indexBuffer = buffer.asIntBuffer( );
    private Random rand = new Random( );
    public static void main (String [] argv)
        throws Exception
```

```
{
    boolean writer = false;
    String filename;
    if (argv.length != 2) {
        System.out.println ("Usage: [ -r | -w ] filename");
        return;
    }
    writer = argv [0].equals ("-w");
    filename = argv [1];
    RandomAccessFile raf = new RandomAccessFile (filename,
        (writer) ? "rw" : "r");
    FileChannel fc = raf.getChannel( );
    LockTest lockTest = new LockTest( );
    if (writer) {
        lockTest.doUpdates (fc);
    } else {
        lockTest.doQueries (fc);
    }
}
// ----------------------------------------------------------------
// Simulate a series of read-only queries while
// holding a shared lock on the index area
void doQueries (FileChannel fc)
    throws Exception
{
    while (true) {
        println ("trying for shared lock...");
        FileLock lock = fc.lock (INDEX_START, INDEX_SIZE, true);
        int reps = rand.nextInt (60) + 20;
        for (int i = 0; i < reps; i++) {
            int n = rand.nextInt (INDEX_COUNT);
            int position = INDEX_START + (n * SIZEOF_INT);
            buffer.clear( );
            fc.read (buffer, position);
            int value = indexBuffer.get (n);
            println ("Index entry " + n + "=" + value);
            // Pretend to be doing some work
            Thread.sleep (100);
```

```
        }
        lock.release( );
        println ("<sleeping>");
        Thread.sleep (rand.nextInt (3000) + 500);
    }
}
// Simulate a series of updates to the index area
// while holding an exclusive lock
void doUpdates (FileChannel fc)
    throws Exception
{
    while (true) {
        println ("trying for exclusive lock...");
        FileLock lock = fc.lock (INDEX_START,
        INDEX_SIZE, false);
        updateIndex (fc);
        lock.release( );
        println ("<sleeping>");
        Thread.sleep (rand.nextInt (2000) + 500);
    }
}
// Write new values to the index slots
private int idxval = 1;
private void updateIndex (FileChannel fc)
    throws Exception
{
    // "indexBuffer" is an int view of "buffer"
    indexBuffer.clear( );
    for (int i = 0; i < INDEX_COUNT; i++) {
        idxval++;
        println ("Updating index " + i + "=" + idxval);
        indexBuffer.put (idxval);
        // Pretend that this is really hard work
        Thread.sleep (500);
    }
    // leaves position and limit correct for whole buffer
    buffer.clear( );
    fc.write (buffer, INDEX_START);
```

```
    }
    // ----------------------------------------------------------------
    private int lastLineLen = 0;
    // Specialized println that repaints the current line
    private void println (String msg)
    {
        System.out.print ("\r ");
        System.out.print (msg);
        for (int i = msg.length( ); i < lastLineLen; i++) {
            System.out.print (" ");
        }
        System.out.print ("\r");
        System.out.flush( );
        lastLineLen = msg.length( );
    }
}
```

例 3-3 共享锁同独占锁交互

以上代码直接忽略了我之前说给的用 `try/catch/finally` 来释放锁的建议，在您自己所写的实际代码中请不要这么懒。

3.4 内存映射文件

新的 *FileChannel* 类提供了一个名为 *map( )*的方法，该方法可以在一个打开的文件和一个特殊类型的 *ByteBuffer* 之间建立一个虚拟内存映射（第一章中已经归纳了什么是内存映射文件以及它们如何同虚拟内存交互）。在 *FileChannel* 上调用 *map( )*方法会创建一个由磁盘文件支持的虚拟内存映射（virtual memory mapping）并在那块虚拟内存空间外部封装一个 *MappedByteBuffer* 对象（参见图 1-6）。

由 *map( )*方法返回的 *MappedByteBuffer* 对象的行为在多数方面类似一个基于内存的缓冲区，只不过该对象的数据元素存储在磁盘上的一个文件中。调用 *get( )*方法会从磁盘文件中获取数据，此数据反映该文件的当前内容，即使在映射建立之后文件已经被一个外部进程做了修改。通过文件映射看到的数据同您用常规方法读取文件看到的内容是完全一样的。相似地，对映射的缓冲区实现一个 *put( )*会更新磁盘上的那个文件（假设对该文件您有写的权限），并且您做的修改对于该文件的其他阅读者也是可见的。

通过内存映射机制来访问一个文件会比使用常规方法读写高效得多，甚至比使用通道的效率都

高。因为不需要做明确的系统调用，那会很消耗时间。更重要的是，操作系统的虚拟内存可以自动缓存内存页（memory page）。这些页是用系统内存来缓存的，所以不会消耗 Java 虚拟机内存堆（memory heap）。

一旦一个内存页已经生效（从磁盘上缓存进来），它就能以完全的硬件速度再次被访问而不需要再次调用系统命令来获取数据。那些包含索引以及其他需频繁引用或更新的内容的巨大而结构化文件能因内存映射机制受益非常多。如果同时结合文件锁定来保护关键区域和控制事务原子性，那您将能了解到内存映射缓冲区如何可以被很好地利用。

下面让我们来看一下如何使用内存映射：

```
public abstract class FileChannel
    extends AbstractChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
    public abstract MappedByteBuffer map (MapMode mode, long position,long size)
    public static class MapMode
    {
        public static final MapMode READ_ONLY
        public static final MapMode READ_WRITE
        public static final MapMode PRIVATE
    }
}
```

可以看到，只有一种 *map( )*方法来创建一个文件映射。它的参数有 mode，position 和 size。参数 `position` 和 `size` 同 *lock( )*方法的这两个参数是一样的（在前面的章节中已有讨论）。我们可以创建一个 *MappedByteBuffer* 来代表一个文件中字节的某个子范围。例如，要映射 100 到 299（包含 299）位置的字节，可以使用下面的代码：

```
buffer = fileChannel.map (FileChannel.MapMode.READ_ONLY, 100, 200);
```

如果要映射整个文件则使用：

```
buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());
```

与文件锁的范围机制不一样，映射文件的范围不应超过文件的实际大小。如果您请求一个超出文件大小的映射，文件会被增大以匹配映射的大小。假如您给 `size` 参数传递的值是

Integer.MAX_VALUE，文件大小的值会膨胀到超过 2.1GB。即使您请求的是一个只读映射，*map( )*方法也会尝试这样做并且大多数情况下都会抛出一个 *IOException* 异常，因为底层的文件不能被修改。该行为同之前讨论的文件“空洞”的行为是一致的。详情请参考 3.3.1 节。

FileChannel 类定义了代表映射模式的常量，且是使用一个类型安全的枚举而非数字值来定义这些常量。这些常量是 *FileChannel* 内部定义的一个内部类（inner class）的静态字段，它们可以在编译时被检查类型，不过您可以像使用一个数值型常量那样使用它们。

同常规的文件句柄类似，文件映射可以是可写的或只读的。前两种映射模式 MapMode.READ_ONLY 和 MapMode.READ_WRITE 意义是很明显的，它们表示您希望获取的映射只读还是允许修改映射的文件。请求的映射模式将受被调用 *map( )*方法的 *FileChannel* 对象的访问权限所限制。如果通道是以只读的权限打开的而您却请求 MapMode.READ_WRITE 模式，那么 *map( )*方法会抛出一个 *NonWritableChannelException* 异常；如果您在一个没有读权限的通道上请求 MapMode.READ_ONLY 映射模式，那么将产生 *NonReadableChannelException* 异常。不过在以 read/write 权限打开的通道上请求一个 MapMode.READ_ONLY 映射却是允许的。*MappedByteBuffer* 对象的可变性可以通过对它调用 *isReadOnly( )*方法来检查。

第三种模式 MapMode.PRIVATE 表示您想要一个写时拷贝（copy-on-write）的映射。这意味着您通过 *put( )*方法所做的任何修改都会导致产生一个私有的数据拷贝并且该拷贝中的数据只有 *MappedByteBuffer* 实例可以看到。该过程不会对底层文件做任何修改，而且一旦缓冲区被施以垃圾收集动作（garbage collected），那些修改都会丢失。尽管写时拷贝的映射可以防止底层文件被修改，您也必须以 read/write 权限来打开文件以建立 MapMode.PRIVATE 映射。只有这样，返回的 M*appedByteBuffer* 对象才能允许使用 *put( )*方法。

写时拷贝这一技术经常被操作系统使用，以在一个进程生成另一个进程时管理虚拟地址空间（virtual address spaces）。使用写时拷贝可以允许父进程和子进程共享内存页直到它们中的一方实际发生修改行为。在处理同一文件的多个映射时也有相同的优势（当然，这需要底层操作系统的支持）。假设一个文件被多个 *MappedByteBuffer* 对象映射并且每个映射都是 MapMode.PRIVATE 模式，那么这份文件的大部分内容都可以被所有映射共享。

选择使用 MapMode.PRIVATE 模式并不会导致您的缓冲区看不到通过其他方式对文件所做的修改。对文件某个区域的修改在使用 MapMode.PRIVATE 模式的缓冲区中都能反映出来，除非该缓冲区已经修改了文件上的同一个区域。正如第一章中所描述的，内存和文件系统都被划分成了页。当在一个写时拷贝的缓冲区上调用 *put( )*方法时，受影响的页会被拷贝，然后更改就会应用到该拷贝中。具体的页面大小取决于具体实现，不过通常都是和底层文件系统的页面大小时一样的。如果缓冲区还没对某个页做出修改，那么这个页就会反映被映射文件的相应位置上的内容。一旦某个页因为写操作而被拷贝，之后就将使用该拷贝页，并且不能被其他缓冲区或文件更新所修改。例 3-5 的代码诠释了这一行为。

您应该注意到了没有 *unmap( )*方法。也就是说，一个映射一旦建立之后将保持有效，直到 *MappedByteBuffer* 对象被施以垃圾收集动作为止。同锁不一样的是，映射缓冲区没有绑定到创建它们的通道上。关闭相关联的 *FileChannel* 不会破坏映射，只有丢弃缓冲区对象本身才会破坏该映射。NIO 设计师们之所以做这样的决定是因为当关闭通道时破坏映射会引起安全问题，而解决该安全问题又会导致性能问题。如果您确实需要知道一个映射是什么时候被破坏的，他们建议使用虚引用（phantom references，参见 *java.lang.ref.PhantomReference*）和一个 cleanup 线程。不过有此需要的概率是微乎其微的。

MemoryMappedBuffer 直接反映它所关联的磁盘文件。如果映射有效时文件被在结构上修改，就会产生奇怪的行为（当然具体的行为是取决于操作系统和文件系统的）。*MemoryMappedBuffer* 有固定的大小，不过它所映射的文件却是弹性的。具体来说，如果映射有效时文件大小变化了，那么缓冲区的部分或全部内容都可能无法访问，并将返回未定义的数据或者抛出未检查的异常。关于被内存映射的文件如何受其他线程或外部进程控制这一点，请务必小心对待。

所有的 *MappedByteBuffer* 对象都是直接的，这意味着它们占用的内存空间位于 Java 虚拟机内存堆之外（并且可能不会算作 Java 虚拟机的内存占用，不过这取决于操作系统的虚拟内存模型）。

因为 *MappedByteBuffers* 也是 *ByteBuffers*，所以能够被传递 *SocketChannel* 之类通道的 *read( )*或 *write( )*以有效传输数据给被映射的文件或从被映射的文件读取数据。如能再结合 scatter/gather，那么从内存缓冲区和被映射文件内容中组织数据就变得很容易了。例 3-4 就是以此方式写 HTTP 回应的。3.4.1 节中将描述一个传输数据给通道或从其他通道读取数据的更加有效的方式。

到现在为止，我们已经讨论完了映射缓冲区同其他缓冲区相同的特性，这些也是您会用得最多的。不过 *MappedByteBuffer* 还定义了几个它独有的方法：

```
public abstract class MappedByteBuffer
    extends ByteBuffer
{
    // This is a partial API listing
    public final MappedByteBuffer load( )
    public final boolean isLoaded( )
    public final MappedByteBuffer force( )
}
```

当我们为一个文件建立虚拟内存映射之后，文件数据通常不会因此被从磁盘读取到内存（这取决于操作系统）。该过程类似打开一个文件：文件先被定位，然后一个文件句柄会被创建，当您准备好之后就可以通过这个句柄来访问文件数据。对于映射缓冲区，虚拟内存系统将根据您的需要来

把文件中相应区块的数据读进来。这个页验证或防错过程需要一定的时间，因为将文件数据读取到内存需要一次或多次的磁盘访问。某些场景下，您可能想先把所有的页都读进内存以实现最小的缓冲区访问延迟。如果文件的所有页都是常驻内存的，那么它的访问速度就和访问一个基于内存的缓冲区一样了。

*load( )*方法会加载整个文件以使它常驻内存。正如我们在第一章所讨论的，一个内存映射缓冲区会建立与某个文件的虚拟内存映射。此映射使得操作系统的底层虚拟内存子系统可以根据需要将文件中相应区块的数据读进内存。已经在内存中或通过验证的页会占用实际内存空间，并且在它们被读进 RAM 时会挤出最近较少使用的其他内存页。

在一个映射缓冲区上调用 *load( )*方法会是一个代价高的操作，因为它会导致大量的页调入（page-in），具体数量取决于文件中被映射区域的实际大小。然而，*load( )*方法返回并不能保证文件就会完全常驻内存，这是由于请求页面调入（demand paging）是动态的。具体结果会因某些因素而有所差异，这些因素包括：操作系统、文件系统，可用 Java 虚拟机内存，最大 Java 虚拟机内存，垃圾收集器实现过程等等。请小心使用 *load( )*方法，它可能会导致您不希望出现的结果。该方法的主要作用是为提前加载文件埋单，以便后续的访问速度可以尽可能的快。

对于那些要求近乎实时访问（near-realtime access）的程序，解决方案就是预加载。但是请记住，不能保证全部页都会常驻内存，不管怎样，之后可能还会有页调入发生。内存页什么时候以及怎样消失受多个因素影响，这些因素中的许多都是不受 Java 虚拟机控制的。JDK 1.4 的 NIO 并没有提供一个可以把页面固定到物理内存上的 API，尽管一些操作系统是支持这样做的。

对于大多数程序，特别是交互性的或其他事件驱动（event-driven）的程序而言，为提前加载文件消耗资源是不划算的。在实际访问时分摊页调入开销才是更好的选择。让操作系统根据需要来调入页意味着不访问的页永远不需要被加载。同预加载整个被映射的文件相比，这很容易减少 I/O 活动总次数。操作系统已经有一个复杂的内存管理系统了，就让它来替您完成此工作吧！

我们可以通过调用 *isLoaded( )*方法来判断一个被映射的文件是否完全常驻内存了。如果该方法返回 `true` 值，那么很大概率是映射缓冲区的访问延迟很少或者根本没有延迟。不过，这也是不能保证的。同样地，返回 `false` 值并不一定意味着访问缓冲区将很慢或者该文件并未完全常驻内存。*isLoaded( )*方法的返回值只是一个暗示，由于垃圾收集的异步性质、底层操作系统以及运行系统的动态性等因素，想要在任意时刻准确判断全部映射页的状态是不可能的。

上面代码中列出的最后一个方法 *force( )*同 *FileChannel* 类中的同名方法相似（参见 3.3.1 节）该方法会强制将映射缓冲区上的更改应用到永久磁盘存储器上。当用 *MappedByteBuffer* 对象来更新一个文件，您应该总是使用 *MappedByteBuffer.force( )*而非 *FileChannel.force( )*，因为通道对象可能不清楚通过映射缓冲区做出的文件的全部更改。*MappedByteBuffer* 没有不更新文件元数据的选项——元数据总是会同时被更新的。请注意，非本地文件系统也同样影响 *MappedByteBuffer.force( )*方法，正如它会对 *FileChannel.force( )*方法有影响，在这里（参见 3.3.1 节）。

如果映射是以 MapMode.READ_ONLY 或 MAP_MODE.PRIVATE 模式建立的，那么调用 *force( )* 方法将不起任何作用，因为永远不会有更改需要应用到磁盘上（但是这样做也是没有害处的）。

例 3-4 诠释了内存映射缓冲区如何同 scatter/gather 结合使用。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileChannel.MapMode;
import java.nio.channels.GatheringByteChannel;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.net.URLConnection;
/**
 * Dummy HTTP server using MappedByteBuffers.
 * Given a filename on the command line, pretend to be
 * a web server and generate an HTTP response containing
 * the file content preceded by appropriate headers. The
 * data is sent with a gathering write.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class MappedHttp
{
    private static final String OUTPUT_FILE = "MappedHttp.out";
    private static final String LINE_SEP = "\r\n";
    private static final String SERVER_ID = "Server: Ronsoft Dummy Server";
    private static final String HTTP_HDR =
        "HTTP/1.0 200 OK" + LINE_SEP + SERVER_ID + LINE_SEP;
    private static final String HTTP_404_HDR =
        "HTTP/1.0 404 Not Found" + LINE_SEP + SERVER_ID + LINE_SEP;
    private static final String MSG_404 = "Could not open file: ";
    public static void main (String [] argv)
        throws Exception
    {
        if (argv.length < 1) {
```

```
    System.err.println ("Usage: filename");
    return;
}
String file = argv [0];
ByteBuffer header = ByteBuffer.wrap (bytes (HTTP_HDR));
ByteBuffer dynhdrs = ByteBuffer.allocate (128);
ByteBuffer [] gather = { header, dynhdrs, null };
String contentType = "unknown/unknown";
long contentLength = -1;
try {
    FileInputStream fis = new FileInputStream (file);
    FileChannel fc = fis.getChannel( );
    MappedByteBuffer filedata =
        fc.map (MapMode.READ_ONLY, 0, fc.size( ));
    gather [2] = filedata;
    contentLength = fc.size( );
    contentType = URLConnection.guessContentTypeFromName (file);
} catch (IOException e) {
    // file could not be opened; report problem
    ByteBuffer buf = ByteBuffer.allocate (128);
    String msg = MSG_404 + e + LINE_SEP;
    buf.put (bytes (msg));
    buf.flip( );
    // Use the HTTP error response
    gather [0] = ByteBuffer.wrap (bytes (HTTP_404_HDR));
    gather [2] = buf;
    contentLength = msg.length( );
    contentType = "text/plain";
}
StringBuffer sb = new StringBuffer( );
sb.append ("Content-Length: " + contentLength);
sb.append (LINE_SEP);
sb.append ("Content-Type: ").append (contentType);
sb.append (LINE_SEP).append (LINE_SEP);
dynhdrs.put (bytes (sb.toString( )));
dynhdrs.flip( );
FileOutputStream fos = new FileOutputStream (OUTPUT_FILE);
FileChannel out = fos.getChannel( );
```

```
        // All the buffers have been prepared; write 'em out
        while (out.write (gather) > 0) {
            // Empty body; loop until all buffers are empty
        }
        out.close( );
        System.out.println ("output written to " + OUTPUT_FILE);
    }
    // Convert a string to its constituent bytes
    // from the ASCII character set
    private static byte [] bytes (String string)
        throws Exception
    {
        return (string.getBytes ("US-ASCII"));
    }
}
```

例 3-4 使用映射文件和 gathering 写操作来编写 HTTP 回复

例 3-5 诠释了各种模式的内存映射如何交互。具体来说，例中代码诠释了写时拷贝是如何页导向（page-oriented）的。当在使用 `MAP_MODE.PRIVATE` 模式创建的 `MappedByteBuffer` 对象上调用 *put( )*方法而引发更改时，就会生成一个受影响页的拷贝。这份私有的拷贝不仅反映本地更改，而且使缓冲区免受来自外部对原来页更改的影响。然而，对于被映射文件其他区域的更改还是可以看到的。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.io.File;
import java.io.RandomAccessFile;
/**
 * Test behavior of Memory mapped buffer types. Create a file, write
 * some data to it, then create three different types of mappings
 * to it. Observe the effects of changes through the buffer APIs
 * and updating the file directly. The data spans page boundaries
 * to illustrate the page-oriented nature of Copy-On-Write mappings.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
```

```
public class MapFile
{
    public static void main (String [] argv)
        throws Exception
    {
        // Create a temp file and get a channel connected to it
        File tempFile = File.createTempFile ("mmaptest", null);
        RandomAccessFile file = new RandomAccessFile (tempFile, "rw");
        FileChannel channel = file.getChannel( );
        ByteBuffer temp = ByteBuffer.allocate (100);
        // Put something in the file, starting at location 0
        temp.put ("This is the file content".getBytes( ));
        temp.flip( );
        channel.write (temp, 0);
        // Put something else in the file, starting at location 8192.
        // 8192 is 8 KB, almost certainly a different memory/FS page.
        // This may cause a file hole, depending on the
        // filesystem page size.
        temp.clear( );
        temp.put ("This is more file content".getBytes( ));
        temp.flip( );
        channel.write (temp, 8192);
        // Create three types of mappings to the same file
        MappedByteBuffer ro = channel.map (
            FileChannel.MapMode.READ_ONLY, 0, channel.size( ));
        MappedByteBuffer rw = channel.map (
            FileChannel.MapMode.READ_WRITE, 0, channel.size( ));
        MappedByteBuffer cow = channel.map (
            FileChannel.MapMode.PRIVATE, 0, channel.size( ));
        // the buffer states before any modifications
        System.out.println ("Begin");
        showBuffers (ro, rw, cow);
        // Modify the copy-on-write buffer
        cow.position (8);
        cow.put ("COW".getBytes( ));
        System.out.println ("Change to COW buffer");
        showBuffers (ro, rw, cow);
        // Modify the read/write buffer
```

```
    rw.position (9);
    rw.put (" R/W ".getBytes( ));
    rw.position (8194);
    rw.put (" R/W ".getBytes( ));
    rw.force( );
    System.out.println ("Change to R/W buffer");
    showBuffers (ro, rw, cow);
    // Write to the file through the channel; hit both pages
    temp.clear( );
    temp.put ("Channel write ".getBytes( ));
    temp.flip( );
    channel.write (temp, 0);
    temp.rewind( );
    channel.write (temp, 8202);
    System.out.println ("Write on channel");
    showBuffers (ro, rw, cow);
    // Modify the copy-on-write buffer again
    cow.position (8207);
    cow.put (" COW2 ".getBytes( ));
    System.out.println ("Second change to COW buffer");
    showBuffers (ro, rw, cow);
    // Modify the read/write buffer
    rw.position (0);
    rw.put (" R/W2 ".getBytes( ));
    rw.position (8210);
    rw.put (" R/W2 ".getBytes( ));
    rw.force( );
    System.out.println ("Second change to R/W buffer");
    showBuffers (ro, rw, cow);
    // cleanup
    channel.close( );
    file.close( );
    tempFile.delete( );
}
// Show the current content of the three buffers
public static void showBuffers (ByteBuffer ro, ByteBuffer rw,
    ByteBuffer cow)
    throws Exception
```

```
    {
        dumpBuffer ("R/O", ro);
        dumpBuffer ("R/W", rw);
        dumpBuffer ("COW", cow);
        System.out.println ("");
    }
    // Dump buffer content, counting and skipping nulls
    public static void dumpBuffer (String prefix, ByteBuffer buffer)
        throws Exception
    {
        System.out.print (prefix + ": '");
        int nulls = 0;
        int limit = buffer.limit( );
        for (int i = 0; i < limit; i++) {
            char c = (char) buffer.get (i);
            if (c == '\u0000') {
                nulls++;
                continue;
            }
            if (nulls != 0) {
                System.out.print ("|[" + nulls
                    + " nulls]|");
                nulls = 0;
            }
            System.out.print (c);
        }
        System.out.println ("'");
    }
}
```

例 3-5 三种类型的内存映射缓冲区

以下是运行上面程序的输出：

```
Begin
R/O: 'This is the file content|[8168 nulls]|This is more file content'
R/W: 'This is the file content|[8168 nulls]|This is more file content'
COW: 'This is the file content|[8168 nulls]|This is more file content'
```

```
Change to COW buffer
R/O: 'This is the file content|[8168 nulls]|This is more file content'
R/W: 'This is the file content|[8168 nulls]|This is more file content'
COW: 'This is COW file content|[8168 nulls]|This is more file content'

Change to R/W buffer
R/O: 'This is t R/W le content|[8168 nulls]|Th R/W more file content'
R/W: 'This is t R/W le content|[8168 nulls]|Th R/W more file content'
COW: 'This is COW file content|[8168 nulls]|Th R/W more file content'

Write on channel
R/O: 'Channel write le content|[8168 nulls]|Th R/W moChannel write t'
R/W: 'Channel write le content|[8168 nulls]|Th R/W moChannel write t'
COW: 'This is COW file content|[8168 nulls]|Th R/W moChannel write t'

Second change to COW buffer
R/O: 'Channel write le content|[8168 nulls]|Th R/W moChannel write t'
R/W: 'Channel write le content|[8168 nulls]|Th R/W moChannel write t'
COW: 'This is COW file content|[8168 nulls]|Th R/W moChann COW2 te t'

Second change to R/W buffer
R/O: ' R/W2 l write le content|[8168 nulls]|Th R/W moChannel R/W2 t'
R/W: ' R/W2 l write le content|[8168 nulls]|Th R/W moChannel R/W2 t'
COW: 'This is COW file content|[8168 nulls]|Th R/W moChann COW2 te t'
```

3.4.1 Channel-to-Channel 传输

由于经常需要从一个位置将文件数据批量传输到另一个位置，*FileChannel* 类添加了一些优化方法来提高该传输过程的效率：

```
public abstract class FileChannel
    extends AbstractChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    // This is a partial API listing
    public abstract long transferTo (long position, long count,
        WritableByteChannel target)
    public abstract long transferFrom (ReadableByteChannel src,
```

```
        long position, long count)
}
```

*transferTo( )*和 *transferFrom( )*方法允许将一个通道交叉连接到另一个通道，而不需要通过一个中间缓冲区来传递数据。只有 *FileChannel* 类有这两个方法，因此 channel-to-channel 传输中通道之一必须是 *FileChannel*。您不能在 socket 通道之间直接传输数据，不过 socket 通道实现 *WritableByteChannel* 和 *ReadableByteChannel* 接口，因此文件的内容可以用 *transferTo( )*方法传输给一个 socket 通道，或者也可以用 *transferFrom( )*方法将数据从一个 socket 通道直接读取到一个文件中。

直接的通道传输不会更新与某个 *FileChannel* 关联的 position 值。请求的数据传输将从 `position` 参数指定的位置开始，传输的字节数不超过 `count` 参数的值。实际传输的字节数会由方法返回，可能少于您请求的字节数。

对于传输数据来源是一个文件的 *transferTo( )*方法，如果 `position + count` 的值大于文件的 size 值，传输会在文件尾的位置终止。假如传输的目的地是一个非阻塞模式的 socket 通道，那么当发送队列（send queue）满了之后传输就可能终止，并且如果输出队列（output queue）已满的话可能不会发送任何数据。类似地，对于 *transferFrom( )*方法：如果来源 `src` 是另外一个 *FileChannel* 并且已经到达文件尾，那么传输将提早终止；如果来源 `src` 是一个非阻塞 socket 通道，只有当前处于队列中的数据才会被传输（可能没有数据）。由于网络数据传输的非确定性，阻塞模式的 socket 也可能会执行部分传输，这取决于操作系统。许多通道实现都是提供它们当前队列中已有的数据而不是等待您请求的全部数据都准备好。

此外，请记住：如果传输过程中出现问题，这些方法也可能抛出 *java.io.IOException* 异常。

Channel-to-channel 传输是可以极其快速的，特别是在底层操作系统提供本地支持的时候。某些操作系统可以不必通过用户空间传递数据而进行直接的数据传输。对于大量的数据传输，这会是一个巨大的帮助（参见例 3-6）。

```
package com.ronsoft.books.nio.channels;
import java.nio.channels.FileChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.FileInputStream;
/**
 * Test channel transfer. This is a very simplistic concatenation
 * program. It takes a list of file names as arguments, opens each
 * in turn and transfers (copies) their content to the given
 * WritableByteChannel (in this case, stdout).
 *
```

```
 * Created April 2002
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class ChannelTransfer
{
    public static void main (String [] argv)
        throws Exception
    {
        if (argv.length == 0) {
            System.err.println ("Usage: filename ...");
            return;
        }
        catFiles (Channels.newChannel (System.out), argv);
    }
    // Concatenate the content of each of the named files to
    // the given channel. A very dumb version of 'cat'.
    private static void catFiles (WritableByteChannel target,
        String [] files)
        throws Exception
    {
        for (int i = 0; i < files.length; i++) {
            FileInputStream fis = new FileInputStream (files [i]);
            FileChannel channel = fis.getChannel( );
            channel.transferTo (0, channel.size( ), target);
            channel.close( );
            fis.close( );
        }
    }
}
```

例 3-6 使用通道传输进行文件连结

3.5 Socket 通道

现在让我们来学习模拟网络套接字的通道类。Socket 通道有与文件通道不同的特征。

新的 socket 通道类可以运行非阻塞模式并且是可选择的。这两个性能可以激活大程序（如网络服务器和中间件组件）巨大的可伸缩性和灵活性。本节中我们会看到，再也没有为每个 socket 连接

使用一个线程的必要了，也避免了管理大量线程所需的上下文交换总开销。借助新的 NIO 类，一个或几个线程就可以管理成百上千的活动 socket 连接了并且只有很少甚至可能没有性能损失。

从图 3-9 可知，全部 socket 通道类（*DatagramChannel*、*SocketChannel* 和 *ServerSocketChannel*）都是由位于 `java.nio.channels.spi` 包中的 *AbstractSelectableChannel* 引申而来。这意味着我们可以用一个 *Selector* 对象来执行 socket 通道的有条件的选择（readiness selection）。选择和多路复用 I/O 会在第四章中讨论。

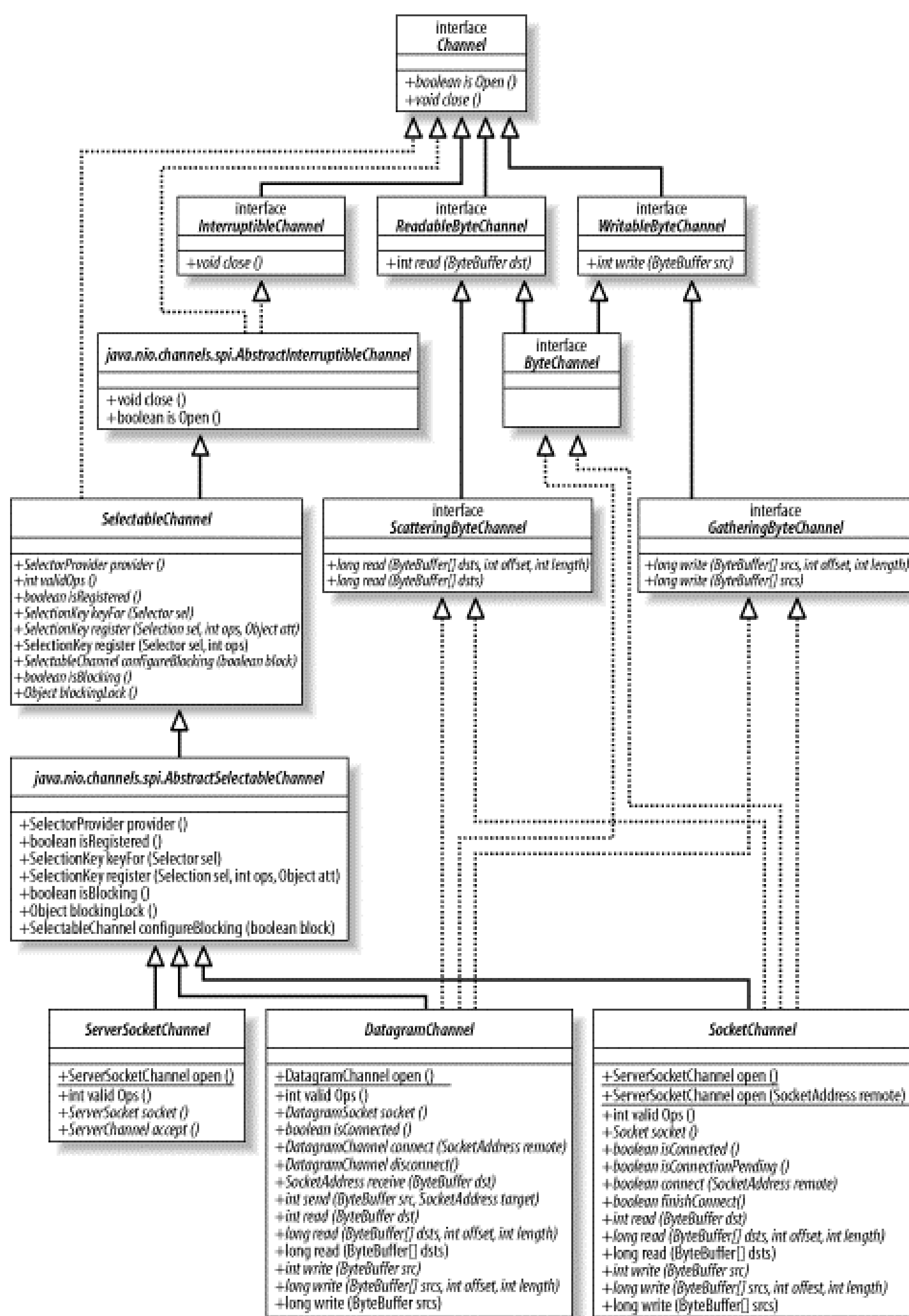


图 3-9 socket 通道类层次结构

请注意 *DatagramChannel* 和 *SocketChannel* 实现定义读和写功能的接口而 *ServerSocketChannel* 不实现。*ServerSocketChannel* 负责监听传入的连接和创建新的 *SocketChannel* 对象，它本身从不传输数据。

在我们具体讨论每一种 socket 通道前，您应该了解 socket 和 socket 通道之间的关系。之前的章节中有写道，通道是一个连接 I/O 服务导管并提供与该服务交互的方法。就某个 socket 而言，它不会再次实现与之对应的 socket 通道类中的 socket 协议 API，而 `java.net` 中已经存在的 socket 通道都可以被大多数协议操作重复使用。

全部 socket 通道类（*DatagramChannel*、*SocketChannel* 和 *ServerSocketChannel*）在被实例化时都会创建一个对等 socket 对象。这些是我们所熟悉的来自 `java.net` 的类（*Socket*、*ServerSocket* 和 *DatagramSocket*），它们已经被更新以识别通道。对等 socket 可以通过调用 *socket( )*方法从一个通道上获取。此外，这三个 `java.net` 类现在都有 *getChannel( )*方法。

虽然每个 socket 通道（在 `java.nio.channels` 包中）都有一个关联的 `java.net` socket 对象，却并非所有的 socket 都有一个关联的通道。如果您用传统方式（直接实例化）创建了一个 Socket 对象，它就不会有关联的 *SocketChannel* 并且它的 *getChannel( )*方法将总是返回 `null`。

Socket 通道委派协议操作给对等 socket 对象。如果在通道类中存在似乎重复的 socket 方法，那么将有某个新的或者不同的行为同通道类上的这个方法相关联。

3.5.1 非阻塞模式

Socket 通道可以在非阻塞模式下运行。这个陈述虽然简单却有着深远的含义。传统 Java socket 的阻塞性质曾经是 Java 程序可伸缩性的最重要制约之一。非阻塞 I/O 是许多复杂的、高性能的程序构建的基础。

要把一个 socket 通道置于非阻塞模式，我们要依靠所有 socket 通道类的公有超级类：*SelectableChannel*。下面的方法就是关于通道的阻塞模式的：

```
public abstract class SelectableChannel
    extends AbstractChannel
    implements Channel
{
    // This is a partial API listing
    public abstract void configureBlocking (boolean block)
        throws IOException;
    public abstract boolean isBlocking( );
```

```
    public abstract Object blockingLock( );
}
```

有条件的选择（readiness selection）是一种可以用来查询通道的机制，该查询可以判断通道是否准备好执行一个目标操作，如读或写。非阻塞 I/O 和可选择性是紧密相连的，那也正是管理阻塞模式的 API 代码要在 *SelectableChannel* 超级类中定义的原因。*SelectableChannel* 的剩余 API 将在第四章中讨论。

设置或重新设置一个通道的阻塞模式是很简单的，只要调用 *configureBlocking( )*方法即可，传递参数值为 `true` 则设为阻塞模式，参数值为 `false` 值设为非阻塞模式。真的，就这么简单！您可以通过调用 *isBlocking( )*方法来判断某个 socket 通道当前处于哪种模式：

```
SocketChannel sc = SocketChannel.open( );
sc.configureBlocking (false); // nonblocking
...
if ( ! sc.isBlocking( )) {
        doSomething (cs);
}
```

服务器端的使用经常会考虑到非阻塞 socket 通道，因为它们使同时管理很多 socket 通道变得更容易。但是，在客户端使用一个或几个非阻塞模式的 socket 通道也是有益处的，例如，借助非阻塞 socket 通道，GUI 程序可以专注于用户请求并且同时维护与一个或多个服务器的会话。在很多程序上，非阻塞模式都是有用的。

偶尔地，我们也会需要防止 socket 通道的阻塞模式被更改。API 中有一个 *blockingLock( )*方法，该方法会返回一个非透明的对象引用。返回的对象是通道实现修改阻塞模式时内部使用的。只有拥有此对象的锁的线程才能更改通道的阻塞模式（对象的锁是用同步的 Java 密码获取的，它不同于我们在 3.3 节中介绍的 *lock( )*方法）。对于确保在执行代码的关键部分时 socket 通道的阻塞模式不会改变以及在不影响其他线程的前提下暂时改变阻塞模式来说，这个方法都是非常方便的。

```
Socket socket = null;
Object lockObj = serverChannel.blockingLock( );
// have a handle to the lock object, but haven't locked it yet
// may block here until lock is acquired
synchronize (lockObj)
{
    // This thread now owns the lock; mode can't be changed
    boolean prevState = serverChannel.isBlocking( );
```

```
    serverChannel.configureBlocking (false);
    socket = serverChannel.accept( );
    serverChannel.configureBlocking (prevState);
}
// lock is now released, mode is allowed to change
if (socket != null) {
    doSomethingWithTheSocket (socket);
}
```

3.5.2 ServerSocketChannel

让我们从最简单的 *ServerSocketChannel* 来开始对 socket 通道类的讨论。以下是 *ServerSocketChannel* 的完整 API：

```
public abstract class ServerSocketChannel
    extends AbstractSelectableChannel
{
    public static ServerSocketChannel open( ) throws IOException
    public abstract ServerSocket socket( );
    public abstract ServerSocket accept( ) throws IOException;
    public final int validOps( )
}
```

ServerSocketChannel 是一个基于通道的 socket 监听器。它同我们所熟悉的 *java.net.ServerSocket* 执行相同的基本任务，不过它增加了通道语义，因此能够在非阻塞模式下运行。

用静态的 *open( )*工厂方法创建一个新的 *ServerSocketChannel* 对象，将会返回同一个未绑定的 *java.net.ServerSocket* 关联的通道。该对等 *ServerSocket* 可以通过在返回的 *ServerSocketChannel* 上调用 *socket( )*方法来获取。作为 *ServerSocketChannel* 的对等体被创建的 *ServerSocket* 对象依赖通道实现。这些 socket 关联的 *SocketImpl* 能识别通道。通道不能被封装在随意的 *socket* 对象外面。

由于 *ServerSocketChannel* 没有 *bind( )*方法，因此有必要取出对等的 socket 并使用它来绑定到一个端口以开始监听连接。我们也是使用对等 *ServerSocket* 的 API 来根据需要设置其他的 socket 选项。

```
ServerSocketChannel ssc = ServerSocketChannel.open( );
ServerSocket serverSocket = ssc.socket( );
// Listen on port 1234
```

```
serverSocket.bind (new InetSocketAddress (1234));
```

同它的对等体 *java.net.ServerSocket* 一样，*ServerSocketChannel* 也有 *accept( )*方法。一旦您创建了一个 *ServerSocketChannel* 并用对等 socket 绑定了它，然后您就可以在其中一个上调用 *accept( )*。如果您选择在 *ServerSocket* 上调用 *accept( )*方法，那么它会同任何其他的 *ServerSocket* 表现一样的行为：总是阻塞并返回一个 *java.net.Socket* 对象。如果您选择在 *ServerSocketChannel* 上调用 *accept( )* 方法则会返回 *SocketChannel* 类型的对象，返回的对象能够在非阻塞模式下运行。假设系统已经有一个安全管理器（security manager），两种形式的方法调用都执行相同的安全检查。

如果以非阻塞模式被调用，当没有传入连接在等待时，*ServerSocketChannel.accept( )*会立即返回 `null`。正是这种检查连接而不阻塞的能力实现了可伸缩性并降低了复杂性。可选择性也因此得到实现。我们可以使用一个选择器实例来注册一个 *ServerSocketChannel* 对象以实现新连接到达时自动通知的功能。例 3-7 演示了如何使用一个非阻塞的 `accept( )`方法：

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.net.InetSocketAddress;
/**
 * Test nonblocking accept( ) using ServerSocketChannel.
 * Start this program, then "telnet localhost 1234" to
 * connect to it.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class ChannelAccept
{
    public static final String GREETING = "Hello I must be going.\r\n";
    public static void main (String [] argv)
        throws Exception
    {
        int port = 1234; // default
        if (argv.length > 0) {
            port = Integer.parseInt (argv [0]);
        }
        ByteBuffer buffer = ByteBuffer.wrap (GREETING.getBytes( ));
        ServerSocketChannel ssc = ServerSocketChannel.open( );
```

```
        ssc.socket( ).bind (new InetSocketAddress (port));
        ssc.configureBlocking (false);
        while (true) {
            System.out.println ("Waiting for connections");
            SocketChannel sc = ssc.accept( );
            if (sc == null) {
                // no connections, snooze a while
                Thread.sleep (2000);
            } else {
                System.out.println ("Incoming connection from: "
                    + sc.socket().getRemoteSocketAddress( ));
                buffer.rewind( );
                sc.write (buffer);
                sc.close( );
            }
        }
    }
}
```

例 3-7 使用 ServerSocketChannel 的非阻塞 accept( )方法

前面列出的最后一个方法 *validOps( )*是同选择器一起使用的。关于选择器，我们将在第四章中予以详细讨论并且会介绍到 *validOps( )*方法。

3.5.3 SocketChannel

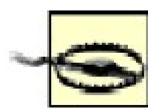
下面开始学习 *SocketChannel*，它是使用最多的 socket 通道类：

```
public abstract class SocketChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
    // This is a partial API listing
    public static SocketChannel open( ) throws IOException
    public static SocketChannel open (InetSocketAddress remote)
    throws IOException
    public abstract Socket socket( );
    public abstract boolean connect (SocketAddress remote)
    throws IOException;
```

```
    public abstract boolean isConnectionPending( );
    public abstract boolean finishConnect( ) throws IOException;
    public abstract boolean isConnected( );
    public final int validOps( )
}
```

Socket 和 *SocketChannel* 类封装点对点、有序的网络连接，类似于我们所熟知并喜爱的 TCP/IP 网络连接。*SocketChannel* 扮演客户端发起同一个监听服务器的连接。直到连接成功，它才能收到数据并且只会从连接到的地址接收。（对于 *ServerSocketChannel*，由于涉及到 *validOps( )*方法，我们将在第四章检查选择器时进行讨论。通用的 read/write 方法也未在此列出，详情请参考 3.1.2 节。）

每个 *SocketChannel* 对象创建时都是同一个对等的 *java.net.Socket* 对象串联的。静态的 *open( )*方法可以创建一个新的 *SocketChannel* 对象，而在新创建的 *SocketChannel* 上调用 *socket( )*方法能返回它对等的 *Socket* 对象；在该 *Socket* 上调用 *getChannel( )*方法则能返回最初的那个 *SocketChannel*。



虽然每个 *SocketChannel* 对象都会创建一个对等的 *Socket* 对象，反过来却不成立。直接创建的 *Socket* 对象不会关联 *SocketChannel* 对象，它们的 *getChannel( )*方法只返回 `null`。

新创建的 *SocketChannel* 虽已打开却是未连接的。在一个未连接的 *SocketChannel* 对象上尝试一个 I/O 操作会导致 *NotYetConnectedException* 异常。我们可以通过在通道上直接调用 *connect( )*方法或在通道关联的 *Socket* 对象上调用 *connect( )*来将该 socket 通道连接。一旦一个 socket 通道被连接，它将保持连接状态直到被关闭。您可以通过调用布尔型的 *isConnected( )*方法来测试某个 *SocketChannel* 当前是否已连接。

第二种带 *InetSocketAddress* 参数形式的 *open( )*是在返回之前进行连接的便捷方法。这段代码：

```
SocketChannel socketChannel =
SocketChannel.open (new InetSocketAddress ("somehost", somePort));
```

等价于下面这段代码：

```
SocketChannel socketChannel = SocketChannel.open( );
socketChannel.connect (new InetSocketAddress ("somehost", somePort));
```

如果您选择使用传统方式进行连接——通过在对等 *Socket* 对象上调用 *connect( )*方法，那么传统的连接语义将适用于此。线程在连接建立好或超时过期之前都将保持阻塞。如果您选择通过在通

道上直接调用 *connect( )*方法来建立连接并且通道处于阻塞模式（默认模式），那么连接过程实际上是一样的。

在 *SocketChannel* 上并没有一种 *connect( )*方法可以让您指定超时（timeout）值，当 *connect( )*方法在非阻塞模式下被调用时 *SocketChannel* 提供并发连接：它发起对请求地址的连接并且立即返回值。如果返回值是 `true`，说明连接立即建立了（这可能是本地环回连接）；如果连接不能立即建立，*connect( )*方法会返回 `false` 且并发地继续连接建立过程。

面向流的的 socket 建立连接状态需要一定的时间，因为两个待连接系统之间必须进行包对话以建立维护流 socket 所需的状态信息。跨越开放互联网连接到远程系统会特别耗时。假如某个 *SocketChannel* 上当前正由一个并发连接，*isConnectPending( )*方法就会返回 `true` 值。

调用 *finishConnect( )*方法来完成连接过程，该方法任何时候都可以安全地进行调用。假如在一个非阻塞模式的 *SocketChannel* 对象上调用 *finishConnect( )*方法，将可能出现下列情形之一：

- *connect( )*方法尚未被调用。那么将产生 *NoConnectionPendingException* 异常。
- 连接建立过程正在进行，尚未完成。那么什么都不会发生，*finishConnect( )*方法会立即返回 `false` 值。
- 在非阻塞模式下调用 *connect( )*方法之后，*SocketChannel* 又被切换回了阻塞模式。那么如果有必要的话，调用线程会阻塞直到连接建立完成，*finishConnect( )*方法接着就会返回 `true` 值。
- 在初次调用 *connect( )*或最后一次调用 *finishConnect( )*之后，连接建立过程已经完成。那么 *SocketChannel* 对象的内部状态将被更新到已连接状态，*finishConnect( )*方法会返回 `true` 值，然后 *SocketChannel* 对象就可以被用来传输数据了。
- 连接已经建立。那么什么都不会发生，*finishConnect( )*方法会返回 `true` 值。

当通道处于中间的连接等待（connection-pending）状态时，您只可以调用 *finishConnect( )*、*isConnectPending( )*或 *isConnected( )*方法。一旦连接建立过程成功完成，*isConnected( )*将返回 `true` 值。

```
InetSocketAddress addr = new InetSocketAddress (host, port);
SocketChannel sc = SocketChannel.open( );
sc.configureBlocking (false);
sc.connect (addr);
while ( ! sc.finishConnect( )) {
    doSomethingElse( );
}
doSomethingWithChannel (sc);
sc.close( );
```

例 3-8 是一段用来管理异步连接的可用代码。

```
package com.ronsoft.books.nio.channels;
import java.nio.channels.SocketChannel;
import java.net.InetSocketAddress;
/**
* Demonstrate asynchronous connection of a SocketChannel.
* @author Ron Hitchens (ron@ronsoft.com)
*/
public class ConnectAsync
{
public static void main (String [] argv) throws Exception
{
String host = "localhost";
int port = 80;
if (argv.length == 2) {
host = argv [0];
port = Integer.parseInt (argv [1]);
}
InetSocketAddress addr = new InetSocketAddress (host, port);
SocketChannel sc = SocketChannel.open( );
sc.configureBlocking (false);
System.out.println ("initiating connection");
sc.connect (addr);
while ( ! sc.finishConnect( )) {
doSomethingUseful( );
}
System.out.println ("connection established");
// Do something with the connected socket
// The SocketChannel is still nonblocking
sc.close( );
}
private static void doSomethingUseful( )
{
System.out.println ("doing something useless");
}
}
```

例 3-8 建立并发连接

如果尝试异步连接失败，那么下次调用 *finishConnect( )*方法会产生一个适当的经检查的异常以指出问题的性质。通道然后就会被关闭并将不能被连接或再次使用。

与连接相关的方法使得我们可以对一个通道进行轮询并在连接进行过程中判断通道所处的状态。第四章中，我们将了解到如何使用选择器来避免进行轮询并在异步连接建立之后收到通知。

Socket 通道是线程安全的。并发访问时无需特别措施来保护发起访问的多个线程，不过任何时候都只有一个读操作和一个写操作在进行中。请记住，sockets 是面向流的而非包导向的。它们可以保证发送的字节会按照顺序到达但无法承诺维持字节分组。某个发送器可能给一个 socket 写入了 20 个字节而接收器调用 *read( )*方法时却只收到了其中的 3 个字节。剩下的 17 个字节还是传输中。由于这个原因，让多个不配合的线程共享某个流 socket 的同一侧绝非一个好的设计选择。

*connect( )*和 *finishConnect( )*方法是互相同步的，并且只要其中一个操作正在进行，任何读或写的方法调用都会阻塞，即使是在非阻塞模式下。如果此情形下您有疑问或不能承受一个读或写操作在某个通道上阻塞，请用 *isConnected( )*方法测试一下连接状态。

3.5.4 DatagramChannel

最后一个 socket 通道是 *DatagramChannel*。正如 *SocketChannel* 对应 *Socket*，*ServerSocketChannel* 对应 *ServerSocket*，每一个 *DatagramChannel* 对象也有一个关联的 *DatagramSocket* 对象。不过原命名模式在此并未适用："DatagramSocketChannel"显得有点笨拙，因此采用了简洁的"DatagramChannel"名称。

正如 *SocketChannel* 模拟连接导向的流协议（如 TCP/IP），*DatagramChannel* 则模拟包导向的无连接协议（如 UDP/IP）：

```
public abstract class DatagramChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
    // This is a partial API listing
    public static DatagramChannel open( ) throws IOException
    public abstract DatagramSocket socket( );
    public abstract DatagramChannel connect (SocketAddress remote)
        throws IOException;
    public abstract boolean isConnected( );
    public abstract DatagramChannel disconnect( ) throws IOException;
    public abstract SocketAddress receive (ByteBuffer dst)
        throws IOException;
```

```
    public abstract int send (ByteBuffer src, SocketAddress target)
    public abstract int read (ByteBuffer dst) throws IOException;
    public abstract long read (ByteBuffer [] dsts) throws IOException;
    public abstract long read (ByteBuffer [] dsts, int offset,
        int length)
        throws IOException;
    public abstract int write (ByteBuffer src) throws IOException;
    public abstract long write(ByteBuffer[] srcs) throws IOException;
    public abstract long write(ByteBuffer[] srcs, int offset,
        int length)
        throws IOException;
}
```

创建 *DatagramChannel* 的模式和创建其他 socket 通道是一样的：调用静态的 *open( )*方法来创建一个新实例。新 *DatagramChannel* 会有一个可以通过调用 *socket( )*方法获取的对等 *DatagramSocket* 对象。*DatagramChannel* 对象既可以充当服务器（监听者）也可以充当客户端（发送者）。如果您希望新创建的通道负责监听，那么通道必须首先被绑定到一个端口或地址/端口组合上。绑定 *DatagramChannel* 同绑定一个常规的 *DatagramSocket* 没什么区别，都是委托对等 socket 对象上的 API 实现的：

```
DatagramChannel channel = DatagramChannel.open( );
DatagramSocket socket = channel.socket( );
socket.bind (new InetSocketAddress (portNumber));
```

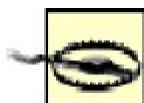
DatagramChannel 是无连接的。每个数据报（datagram）都是一个自包含的实体，拥有它自己的目的地址及不依赖其他数据报的数据净荷。与面向流的的 socket 不同，*DatagramChannel* 可以发送单独的数据报给不同的目的地址。同样，*DatagramChannel* 对象也可以接收来自任意地址的数据包。每个到达的数据报都含有关于它来自何处的信息（源地址）。

一个未绑定的 *DatagramChannel* 仍能接收数据包。当一个底层 socket 被创建时，一个动态生成的端口号就会分配给它。绑定行为要求通道关联的端口被设置为一个特定的值（此过程可能涉及安全检查或其他验证）。不论通道是否绑定，所有发送的包都含有 `DatagramChannel` 的源地址（带端口号）。未绑定的 *DatagramChannel* 可以接收发送给它的端口的包，通常是来回应该通道之前发出的一个包。已绑定的通道接收发送给它们所绑定的熟知端口（wellknown port）的包。数据的实际发送或接收是通过 `send( )`和 `receive( )`方法来实现的：

```
public abstract class DatagramChannel
    extends AbstractSelectableChannel
```

```
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
    // This is a partial API listing
    public abstract SocketAddress receive (ByteBuffer dst)
        throws IOException;
    public abstract int send (ByteBuffer src, SocketAddress target)
}
```

*receive( )*方法将下次将传入的数据报的数据净荷复制到预备好的 ByteBuffer 中并返回一个 *SocketAddress* 对象以指出数据来源。如果通道处于阻塞模式，*receive( )*可能无限期地休眠直到有包到达。如果是非阻塞模式，当没有可接收的包时则会返回 `null`。如果包内的数据超出缓冲区能承受的范围，多出的数据都会被悄悄地丢弃。



假如您提供的 *ByteBuffer* 没有足够的剩余空间来存放您正在接收的数据包，没有被填充的字节都会被悄悄地丢弃。

调用 *send( )*会发送给定 *ByteBuffer* 对象的内容到给定 *SocketAddress* 对象所描述的目的地址和端口，内容范围为从当前 position 开始到末尾处结束。如果 *DatagramChannel* 对象处于阻塞模式，调用线程可能会休眠直到数据报被加入传输队列。如果通道是非阻塞的，返回值要么是字节缓冲区的字节数，要么是“`0`”。发送数据报是一个全有或全无（all-or-nothing）的行为。如果传输队列没有足够空间来承载整个数据报，那么什么内容都不会被发送。

如果安装了安全管理器，那么每次调用 *send( )*或 *receive( )*时安全管理器的 *checkConnect( )*方法都会被调用以验证目的地址，除非通道处于已连接的状态（本节后面会讨论到）。

请注意，数据报协议的不可靠性是固有的，它们不对数据传输做保证。*send( )*方法返回的非零值并不表示数据报到达了目的地，仅代表数据报被成功加到本地网络层的传输队列。此外，传输过程中的协议可能将数据报分解成碎片。例如，以太网不能传输超过 1,500 个字节左右的包。如果您的数据报比较大，那么就会存在被分解成碎片的风险，成倍地增加了传输过程中包丢失的几率。被分解的数据报在目的地会被重新组合起来，接收者将看不到碎片。但是，如果有一个碎片不能按时到达，那么整个数据报将被丢弃。

DatagramChannel 有一个 *connect( )*方法：

```
public abstract class DatagramChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
```

```
{
    // This is a partial API listing
    public abstract DatagramChannel connect (SocketAddress remote)
        throws IOException;
    public abstract boolean isConnected( );
    public abstract DatagramChannel disconnect( ) throws IOException;
}
```

DatagramChannel 对数据报 socket 的连接语义不同于对流 socket 的连接语义。有时候，将数据报对话限制为两方是很可取的。将 *DatagramChannel* 置于已连接的状态可以使除了它所“连接”到的地址之外的任何其他源地址的数据报被忽略。这是很有帮助的，因为不想要的包都已经被网络层丢弃了，从而避免了使用代码来接收、检查然后丢弃包的麻烦。

当 *DatagramChannel* 已连接时，使用同样的令牌，您不可以发送包到除了指定给 *connect( )*方法的目的地址以外的任何其他地址。试图一定要这样做的话会导致一个 *SecurityException* 异常。

我们可以通过调用带 *SocketAddress* 对象的 *connect( )*方法来连接一个 *DatagramChannel*，该 *SocketAddress* 对象描述了 *DatagramChannel* 远程对等体的地址。如果已经安装了一个安全管理器，那么它会进行权限检查。之后，每次 send/receive 时就不会再有安全检查了，因为来自或去到任何其他地址的包都是不允许的。

已连接通道会发挥作用的使用场景之一是一个客户端/服务器模式、使用 UDP 通讯协议的实时游戏。每个客户端都只和同一台服务器进行会话而希望忽视任何其他来源地数据包。将客户端的 *DatagramChannel* 实例置于已连接状态可以减少按包计算的总开销（因为不需要对每个包进行安全检查）和剔除来自欺骗玩家的假包。服务器可能也想要这样做，不过需要每个客户端都有一个 *DatagramChannel* 对象。

不同于流 socket，数据报 socket 的无状态性质不需要同远程系统进行对话来建立连接状态。没有实际的连接，只有用来指定允许的远程地址的本地状态信息。由于此原因，*DatagramChannel* 上也就没有单独的 *finishConnect( )*方法。我们可以使用 *isConnected( )*方法来测试一个数据报通道的连接状态。

不同于 *SocketChannel*（必须连接了才有用并且只能连接一次），*DatagramChannel* 对象可以任意次数地进行连接或断开连接。每次连接都可以到一个不同的远程地址。调用 *disconnect( )*方法可以配置通道，以便它能再次接收来自安全管理器（如果已安装）所允许的任意远程地址的数据或发送数据到这些地址上。

当一个 *DatagramChannel* 处于已连接状态时，发送数据将不用提供目的地址而且接收时的源地址也是已知的。这意味着 *DatagramChannel* 已连接时可以使用常规的 *read( )*和 *write( )*方法，包括

scatter/gather 形式的读写来组合或分拆包的数据：

```
public abstract class DatagramChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
    // This is a partial API listing
    public abstract int read (ByteBuffer dst) throws IOException;
    public abstract long read (ByteBuffer [] dsts) throws IOException;
    public abstract long read (ByteBuffer [] dsts, int offset,
        int length)
        throws IOException;
    public abstract int write (ByteBuffer src) throws IOException;
    public abstract long write(ByteBuffer[] srcs) throws IOException;
    public abstract long write(ByteBuffer[] srcs, int offset,
        int length)
        throws IOException;
}
```

*read( )*方法返回读取字节的数量，如果通道处于非阻塞模式的话这个返回值可能是“0”。*write( )*方法的返回值同 *send( )*方法一致：要么返回您的缓冲区中的字节数量，要么返回“0”（如果由于通道处于非阻塞模式而导致数据报不能被发送）。当通道不是已连接状态时调用 *read( )*或 *write( )*方法，都将产生 *NotYetConnectedException* 异常。

数据报通道不同于流 socket。由于它们的有序而可靠的数据传输特性，流 socket 非常得有用。大多数网络连接都是流 socket（TCP/IP 就是一个显著的例子）。但是，像 TCP/IP 这样面向流的的协议为了在包导向的互联网基础设施上维护流语义必然会产生巨大的开销，并且流隐喻不能适用所有的情形。数据报的吞吐量要比流协议高很多，并且数据报可以做很多流无法完成的事情。

下面列出了一些选择数据报 socket 而非流 socket 的理由：

- 您的程序可以承受数据丢失或无序的数据。
- 您希望“发射后不管”（fire and forget）而不需要知道您发送的包是否已接收。
- 数据吞吐量比可靠性更重要。
- 您需要同时发送数据给多个接受者（多播或者广播）。
- 包隐喻比流隐喻更适合手边的任务。

如果以上特征中的一个或多个适用于您的程序，那么数据报设计对您来说就是合适的。

例 3-9 显示了如何使用 *DatagramChannel* 发送请求到多个地址上的时间服务器。*DatagramChannel* 接着会等待回复（reply）的到达。对于每个返回的回复，远程时间会同本地时间进行比较。由于数据报传输不保证一定成功，有些回复可能永远不会到达。大多数 Linux 和 Unix 系统都默认提供时间服务。互联网上也有一个公共时间服务器，如 time.nist.gov。防火墙或者您的 ISP 可能会干扰数据报传输，这是因人而异的。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.DatagramChannel;
import java.net.InetSocketAddress;
import java.util.Date;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
/**
 * Request time service, per RFC 868. RFC 868
 * (http://www.ietf.org/rfc/rfc0868.txt) is a very simple time protocol
 * whereby one system can request the current time from another system.
 * Most Linux, BSD and Solaris systems provide RFC 868 time service
 * on port 37. This simple program will inter-operate with those.
 * The National Institute of Standards and Technology (NIST) operates
 * a public time server at time.nist.gov.
 *
 * The RFC 868 protocol specifies a 32 bit unsigned value be sent,
 * representing the number of seconds since Jan 1, 1900. The Java
 * epoch begins on Jan 1, 1970 (same as unix) so an adjustment is
 * made by adding or subtracting 2,208,988,800 as appropriate. To
 * avoid shifting and masking, a four-byte slice of an
 * eight-byte buffer is used to send/recieve. But getLong( )
 * is done on the full eight bytes to get a long value.
 *
 * When run, this program will issue time requests to each hostname
 * given on the command line, then enter a loop to receive packets.
 * Note that some requests or replies may be lost, which means
 * this code could block forever.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
```

```
 */
public class TimeClient
{
    private static final int DEFAULT_TIME_PORT = 37;
    private static final long DIFF_1900 = 2208988800L;
    protected int port = DEFAULT_TIME_PORT;
    protected List remoteHosts;
    protected DatagramChannel channel;
    public TimeClient (String [] argv) throws Exception
    {
        if (argv.length == 0) {
            throw new Exception ("Usage: [ -p port ] host ...");
        }
        parseArgs (argv);
        this.channel = DatagramChannel.open( );
    }
    protected InetSocketAddress receivePacket (DatagramChannel channel,
        ByteBuffer buffer)
        throws Exception
    {
        buffer.clear( );
        // Receive an unsigned 32-bit, big-endian value
        return ((InetSocketAddress) channel.receive (buffer));
    }
    // Send time requests to all the supplied hosts
    protected void sendRequests( )
        throws Exception
    {
        ByteBuffer buffer = ByteBuffer.allocate (1);
        Iterator it = remoteHosts.iterator( );
        while (it.hasNext( )) {
            InetSocketAddress sa = (InetSocketAddress) it.next( );
            System.out.println ("Requesting time from "
            + sa.getHostName() + ":" + sa.getPort( ));
            // Make it empty (see RFC868)
            buffer.clear().flip( );
            // Fire and forget
            channel.send (buffer, sa);
```

```
    }
}
// Receive any replies that arrive
public void getReplies( ) throws Exception
{
    // Allocate a buffer to hold a long value
    ByteBuffer longBuffer = ByteBuffer.allocate (8);
    // Assure big-endian (network) byte order
    longBuffer.order (ByteOrder.BIG_ENDIAN);
    // Zero the whole buffer to be sure
    longBuffer.putLong (0, 0);
    // Position to first byte of the low-order 32 bits
    longBuffer.position (4);
    // Slice the buffer; gives view of the low-order 32 bits
    ByteBuffer buffer = longBuffer.slice( );
    int expect = remoteHosts.size( );
    int replies = 0;
    System.out.println ("");
    System.out.println ("Waiting for replies...");
    while (true) {
        InetSocketAddress sa;
        sa = receivePacket (channel, buffer);
        buffer.flip( );
        replies++;
        printTime (longBuffer.getLong (0), sa);
        if (replies == expect) {
            System.out.println ("All packets answered");
            break;
        }
        // Some replies haven't shown up yet
        System.out.println ("Received " + replies
            + " of " + expect + " replies");
    }
}
// Print info about a received time reply
protected void printTime (long remote1900, InetSocketAddress sa)
{
    // local time as seconds since Jan 1, 1970
```

```
    long local = System.currentTimeMillis( ) / 1000;
    // remote time as seconds since Jan 1, 1970
    long remote = remote1900 - DIFF_1900;
    Date remoteDate = new Date (remote * 1000);
    Date localDate = new Date (local * 1000);
    long skew = remote - local;
    System.out.println ("Reply from "
        + sa.getHostName() + ":" + sa.getPort( ));
    System.out.println (" there: " + remoteDate);
    System.out.println (" here: " + localDate);
    System.out.print (" skew: ");
    if (skew == 0) {
        System.out.println ("none");
    } else if (skew > 0) {
        System.out.println (skew + " seconds ahead");
    } else {
        System.out.println ((-skew) + " seconds behind");
    }
}
protected void parseArgs (String [] argv)
{
    remoteHosts = new LinkedList( );
    for (int i = 0; i < argv.length; i++) {
        String arg = argv [i];
        // Send client requests to the given port
        if (arg.equals ("-p")) {
        i++;
        this.port = Integer.parseInt (argv [i]);
        continue;
        }
        // Create an address object for the hostname
        InetSocketAddress sa = new InetSocketAddress (arg, port);
        // Validate that it has an address
        if (sa.getAddress( ) == null) {
        System.out.println ("Cannot resolve address: "
        + arg);
        continue;
        }
```

```
            remoteHosts.add (sa);
        }
    }
    // ---------------------------------------------------------------
    public static void main (String [] argv)
        throws Exception
    {
        TimeClient client = new TimeClient (argv);
        client.sendRequests( );
        client.getReplies( );
    }
}
```

例 3-9 使用 DatagramChannel 的时间服务客户端

例 3-10 中的程序是一个 RFC 868 时间服务器。这段代码回答来自例 3-9 中的客户端的请求并显示出 *DatagramChannel* 是怎样绑定到一个熟知端口然后开始监听来自客户端的请求的。该时间服务器仅监听数据报（UDP）请求。大多数 Unix 和 Linux 系统提供的 *rdate* 命令使用 TCP 协议连接到一个 RFC 868 时间服务。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.DatagramChannel;
import java.net.SocketAddress;
import java.net.InetSocketAddress;
import java.net.SocketException;
/**
 * Provide RFC 868 time service (http://www.ietf.org/rfc/rfc0868.txt).
 * This code implements an RFC 868 listener to provide time
 * service. The defined port for time service is 37. On most
 * unix systems, root privilege is required to bind to ports
 * below 1024. You can either run this code as root or
 * provide another port number on the command line. Use
 * "-p port#" with TimeClient if you choose an alternate port.
 *
 * Note: The familiar rdate command on unix will probably not work
 * with this server. Most versions of rdate use TCP rather than UDP
 * to request the time.
```

```
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class TimeServer
{
    private static final int DEFAULT_TIME_PORT = 37;
    private static final long DIFF_1900 = 2208988800L;
    protected DatagramChannel channel;
    public TimeServer (int port)
        throws Exception
    {
        this.channel = DatagramChannel.open( );
        this.channel.socket( ).bind (new InetSocketAddress (port));
        System.out.println ("Listening on port " + port
            + " for time requests");
    }
    public void listen( ) throws Exception
    {
        // Allocate a buffer to hold a long value
        ByteBuffer longBuffer = ByteBuffer.allocate (8);
        // Assure big-endian (network) byte order
        longBuffer.order (ByteOrder.BIG_ENDIAN);
        // Zero the whole buffer to be sure
        longBuffer.putLong (0, 0);
        // Position to first byte of the low-order 32 bits
        longBuffer.position (4);
        // Slice the buffer; gives view of the low-order 32 bits
        ByteBuffer buffer = longBuffer.slice( );
        while (true) {
            buffer.clear( );
            SocketAddress sa = this.channel.receive (buffer);
            if (sa == null) {
                continue; // defensive programming
            }
            // Ignore content of received datagram per RFC 868
            System.out.println ("Time request from " + sa);
            buffer.clear( ); // sets pos/limit correctly
            // Set 64-bit value; slice buffer sees low 32 bits
```

```
            longBuffer.putLong (0,
                (System.currentTimeMillis( ) / 1000) + DIFF_1900);
            this.channel.send (buffer, sa);
        }
    }
    // ---------------------------------------------------------------
    public static void main (String [] argv)
        throws Exception
    {
        int port = DEFAULT_TIME_PORT;
        if (argv.length > 0) {
            port = Integer.parseInt (argv [0]);
        }
        try {
            TimeServer server = new TimeServer (port);
            server.listen( );
        } catch (SocketException e) {
            System.out.println ("Can't bind to port " + port
                + ", try a different one");
        }
    }
}
```

例 3-10 DatagramChannel 时间服务器

3.6 管道

`java.nio.channels` 包中含有一个名为 *Pipe*（管道）的类。广义上讲，管道就是一个用来在两个实体之间单向传输数据的导管。管道的概念对于 Unix（和类 Unix）操作系统的用户来说早就很熟悉了。Unix 系统中，管道被用来连接一个进程的输出和另一个进程的输入。*Pipe* 类实现一个管道范例，不过它所创建的管道是进程内（在 Java 虚拟机进程内部）而非进程间使用的。参见图 3-10。

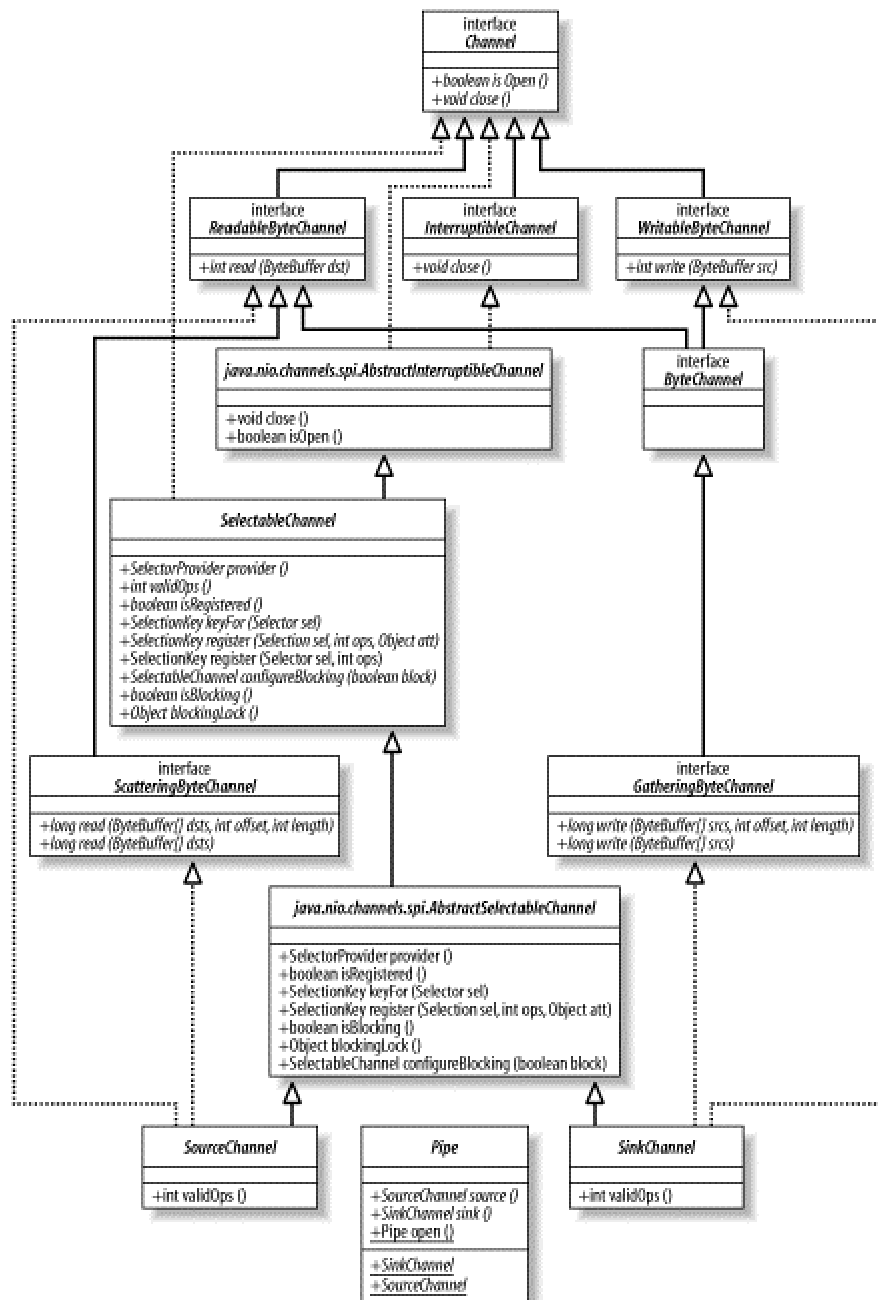


图 3-10 Pipe 类层次结构

Pipe 类创建一对提供环回机制的 *Channel* 对象。这两个通道的远端是连接起来的，以便任何写在 *SinkChannel* 对象上的数据都能出现在 *SourceChannel* 对象上。图 3-11 显示了 `Pipe` 的类层级。

```
package java.nio.channels;
public abstract class Pipe
{
    public static Pipe open( ) throws IOException
    public abstract SourceChannel source( );
    public abstract SinkChannel sink( );
```

```
    public static abstract class SourceChannel
        extends AbstractSelectableChannel
        implements ReadableByteChannel, ScatteringByteChannel
    public static abstract class SinkChannel
        extends AbstractSelectableChannel
        implements WritableByteChannel, GatheringByteChannel
}
```

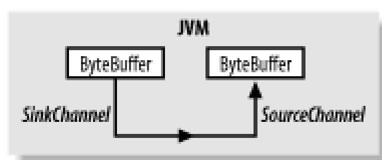


图 3-11 管道是一对循环的通道

Pipe 实例是通过调用不带参数的 *Pipe.open( )*工厂方法来创建的。*Pipe* 类定义了两个嵌套的通道类来实现管路。这两个类是 *Pipe.SourceChannel*（管道负责读的一端）和 *Pipe.SinkChannel*（管道负责写的一端）。这两个通道实例是在 *Pipe* 对象创建的同时被创建的，可以通过在 *Pipe* 对象上分别调用 *source( )*和 *sink( )*方法来取回。

此时，您可能在想管道到底有什么作用。您不能使用 *Pipe* 在操作系统级的进程间建立一个类 Unix 管道（您可以使用 *SocketChannel* 来建立）。*Pipe* 的 source 通道和 sink 通道提供类似 *java.io.PipedInputStream* 和 *java.io.PipedOutputStream* 所提供的功能，不过它们可以执行全部的通道语义。请注意，*SinkChannel* 和 *SourceChannel* 都由 *AbstractSelectableChannel* 引申而来（所以也是从 *SelectableChannel* 引申而来），这意味着 pipe 通道可以同选择器一起使用（参见第四章）。

管道可以被用来仅在同一个 Java 虚拟机内部传输数据。虽然有更加有效率的方式来在线程之间传输数据，但是使用管道的好处在于封装性。生产者线程和用户线程都能被写道通用的 *Channel* API 中。根据给定的通道类型，相同的代码可以被用来写数据到一个文件、socket 或管道。选择器可以被用来检查管道上的数据可用性，如同在 socket 通道上使用那样地简单。这样就可以允许单个用户线程使用一个 *Selector* 来从多个通道有效地收集数据，并可任意结合网络连接或本地工作线程使用。因此，这些对于可伸缩性、冗余度以及可复用性来说无疑都是意义重大的。

Pipes 的另一个有用之处是可以用来辅助测试。一个单元测试框架可以将某个待测试的类连接到管道的“写”端并检查管道的“读”端出来的数据。它也可以将被测试的类置于通道的“读”端并将受控的测试数据写进其中。两种场景对于回归测试都是很有帮助的。

管路所能承载的数据量是依赖实现的（implementation-dependent）。唯一可保证的是写到 *SinkChannel* 中的字节都能按照同样的顺序在 *SourceChannel* 上重现。例 3-11 诠释了如何使用管道。

```
package com.ronsoft.books.nio.channels;
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Pipe;
import java.nio.channels.Channels;
import java.util.Random;
/**
 * Test Pipe objects using a worker thread.
 *
 * Created April, 2002
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class PipeTest
{
    public static void main (String [] argv)
        throws Exception
    {
        // Wrap a channel around stdout
        WritableByteChannel out = Channels.newChannel (System.out);
        // Start worker and get read end of channel
        ReadableByteChannel workerChannel = startWorker (10);
        ByteBuffer buffer = ByteBuffer.allocate (100);
        while (workerChannel.read (buffer) >= 0) {
            buffer.flip( );
            out.write (buffer);
            buffer.clear( );
        }
    }
    // This method could return a SocketChannel or
    // FileChannel instance just as easily
    private static ReadableByteChannel startWorker (int reps)
        throws Exception
    {
        Pipe pipe = Pipe.open( );
        Worker worker = new Worker (pipe.sink( ), reps);
        worker.start( );
        return (pipe.source( ));
```

```
}
// ----------------------------------------------------------------
/**
 * A worker thread object which writes data down a channel.
 * Note: this object knows nothing about Pipe, uses only a
 * generic WritableByteChannel.
 */
private static class Worker extends Thread
{
    WritableByteChannel channel;
    private int reps;
    Worker (WritableByteChannel channel, int reps)
    {
        this.channel = channel;
        this.reps = reps;
    }
    // Thread execution begins here
    public void run( )
    {
        ByteBuffer buffer = ByteBuffer.allocate (100);
        try {
            for (int i = 0; i < this.reps; i++) {
                doSomeWork (buffer);
                // channel may not take it all at once
                while (channel.write (buffer) > 0) {
                    // empty
                }
            }
            this.channel.close( );
        } catch (Exception e) {
            // easy way out; this is demo code
            e.printStackTrace( );
        }
    }
    private String [] products = {
        "No good deed goes unpunished",
        "To be, or what?",
        "No matter where you go, there you are",
```

```
            "Just say \"Yo\"",
            "My karma ran over my dogma"
        };
        private Random rand = new Random( );
        private void doSomeWork (ByteBuffer buffer)
        {
            int product = rand.nextInt (products.length);
            buffer.clear( );
            buffer.put (products [product].getBytes( ));
            buffer.put ("\r\n".getBytes( ));
            buffer.flip( );
        }
    }
}
```

例 3-11 工作线程对一个管道进行写操作

3.7 通道工具类

NIO 通道提供了一个全新的类似流的 I/O 隐喻，但是我们所熟悉的字节流以及字符读写器仍然存在并被广泛使用。通道可能最终会改进加入到 `java.io` 类中（这是一个实现细节），但是 `java.io` 流所代表的 API 和读写器却不会很快消失（它们也不应该消失）。

一个工具类（`java.nio.channels.Channels` 的一个稍微重复的名称）定义了几种静态的工厂方法以使通道可以更加容易地同流和读写器互联。表 3-2 对这些方法做了一个汇总。

表 3-2 java.nio.channels.Channels 工具方法汇总

方法	**返回**	**描述**
newChannel (InputStream in)	*ReadableByteChannel*	返回一个将从给定的输入流读取数据的通道。
newChannel (OutputStream out)	*WritableByteChannel*	返回一个将向给定的输出流写入数据的通道。
newInputStream (ReadableByteChannel ch)	*InputStream*	返回一个将从给定的通道读取字节的流。
newOutputStream (WritableByteChannel ch)	*OutputStream*	返回一个将向给定的通道写入字节的流。
newReader (ReadableByteChannel ch, CharsetDecoder dec, int	*Reader*	返回一个 reader，它将从给定的通道读取字节并依据提供的

minBufferCap)		*CharsetDecoder* 对读取到的字节进行解码。字符集编码/解码将在第六章中讨论。
newReader(ReadableByteChannel ch, String csName)	*Reader*	返回一个 reader，它将从给定的通道读取字节并依据提供的字符集名称将读取到的字节解码成字符。
newWriter (WritableByteChannel ch, CharsetEncoder dec, int minBufferCap)	*Writer*	返回一个 writer，它将使用提供的 *CharsetEncoder* 对象对字符编码并写到给定的通道中。
newWriter (WritableByteChannel ch, String csName)	*Writer*	返回一个 writer，它将依据提供的字符集名称对字符编码并写到给定的通道中。

回忆一下，常规的流仅传输字节，readers 和 writers 则作用于字符数据。表 3-2 的前四行描述了用于连接流、通道的方法。因为流和通道都是运行在字节流基础上的，所以这四个方法直接将流封装在通道上，反之亦然。

Readers 和 Writers 运行在字符的基础上，在 Java 的世界里字符同字节是完全不同的。将一个通道（仅了解字节）连接到一个 reader 或 writer 需要一个中间对话来处理字节/字符（`byte/char`）阻抗失配。为此，表 3-2 的后半部分描述的工厂方法使用了字符集编码器和解码器。字符集以及字符集转码将在第六章中详细讨论。

这些方法返回的包封 *Channel* 对象可能会也可能不会实现 *InterruptibleChannel* 接口，它们也可能不是从 *SelectableChannel* 引申而来。因此，可能无法将这些包封通道同 `java.nio.channels` 包中定义的其他通道类型交换使用。细节是依赖实现的。如果您的程序依赖这些语义，那么请使用操作器实例测试一下返回的通道对象。

3.8 总结

本章中我们讨论了通道的很多方面的内容。通道组成了基础设施或者说管道设施，该设施在操作系统（或通道连接到的任意东西）的 *ByteBuffers* 和 I/O 服务之间传输数据。本章中讨论到的关键概念有：

基本的通道操作

在 3.1 节中我们学习了通道的基本操作，具体包括：怎样使用所有通道都通用的 API 方法调用来打开一个通道以及完成操作时如何关闭通道。

Scatter/Gather 通道

在 3.2 节中我们介绍了如何使用通道来 scatter/gather I/O。矢量化的 I/O 使您可以在多个缓冲区上自动执行一个 I/O 操作。

文件通道

在 3.3 节中我们讨论了多层面的 *FileChannel* 类。这个强大的新通道提供了对高级文件操作的访问，以前这是不对 Java 编程开放的。新的功能特性包括：文件锁定、内存映射文件以及 channel-to-channel 传输。

Socket 通道

在 3.5 节中我们覆盖了几种类型的 socket 通道。同时，我们也讨论了 socket 通道所支持的一个重要新特性——非阻塞模式。

管道

在 3.6 节中我们看了一下 *Pipe* 类，这是一个使用专门的通道实现的新循环机制，非常有用。

通道工具类

通道类中包含了工具方法，这些方法用于交叉连接通道和常规的字节流以及字符读写器对象。参见 3.7 节。

您的 NIO 拨号中有许多通道，现在我们都已经全部学习了。本章中需要吸收的内容很多。通道是 NIO 的一个关键抽象。既然我们了解了通道是什么以及怎样有效地使用它们来访问本地操作系统的 I/O 服务，那么现在是时候前进到 NIO 的下一个主要创新了。下一章中，我们将学习如何简单有效地管理许多这些强大的新通道。

下去冲个澡休息下，再逛下礼品店，然后请重新登上本巴士。我们的下一站是：选择器（Selectors）。

第四章 选择器

生活就是一系列猛然的醒悟。

——R. Van Winkle

在本章中，我们将探索选择器(selectors)。选择器提供选择执行已经就绪的任务的能力，这使得多元 I/O 成为可能。就像在第一章中描述的那样，就绪选择和多元执行使得单线程能够有效率地同时管理多个 I/O 通道(channels)。C/C++代码的工具箱中，许多年前就已经有 *select()*和 *poll()*这两个 POSIX（可移植性操作系统接口）系统调用可供使用了。许过操作系统也提供相似的功能，但对 Java 程序员来说，就绪选择功能直到 JDK 1.4 才成为可行的方案。对于主要的工作经验都是基于 Java 环境的开发的程序员来说，之前可能还没有碰到过这种 I/O 模型。

为了更好地说明就绪选择，让我们回到第三章的带传送通道的银行的例子里。想象一下，一个有三个传送通道的银行。在传统的（非选择器）的场景里，想象一下每个银行的传送通道都有一个气动导管，传送到银行里它对应的出纳员的窗口，并且每一个窗口与其他窗口是用墙壁分隔开的。这意味着每个导管(通道)需要一个专门的出纳员(工作线程)。这种方式不易于扩展，而且也是十分浪费的。对于每个新增加的导管（通道），都需要一个新的出纳员，以及其他相关的经费，如表格、椅子、纸张的夹子（内存、CPU 周期、上下文切换）等等。并且当事情变慢下来时，这些资源（以及相关的花费）大多数时候是闲置的。

现在想象一下另一个不同的场景，每一个气动导管（通道）都只与一个出纳员的窗口连接。这个窗口有三个槽可以放置运输过来的物品（数据缓冲区），每个槽都有一个指示器（选择键，selection key），当运输的物品进入时会亮起。同时想象一下出纳员（工作线程）有一个花尽量多的时间阅读《自己动手编写个人档案》一书的癖好。在每一段的最后，出纳员看一眼指示灯（调用 select( )函数），来决定人一个通道是否已经就绪（就绪选择）。在传送带闲置时，出纳员（工作线程）可以做其他事情，但需要注意的时候又可以进行及时的处理。

虽然这种分析并不精确，但它描述了快速检查大量资源中的任意一个是否需要关注，而在某些东西没有准备好时又不必被迫等待的通用模式。这种检查并继续的能力是可扩展性的关键，它使得仅仅使用单一的线程就可以通过就绪选择来监控大量的通道。

选择器及相关的类就提供了这种 API，使得我们可以在通道上进行就绪选择。

4.1 选择器基础

掌握本章中讨论的主题，在某种程度上，比直接理解缓冲区和通道类更困难一些。这会复杂一些，因为涉及了三个主要的类，它们都会同时参与到整个过程中。如果您发现自己有些困惑，记录下来并先看其他内容。一旦您了解了各个部分是如何相互适应的，以及每个部分扮演的角色，您就会理解这些内容了。

我们会先从总体开始，然后分解为细节。您需要将之前创建的一个或多个可选择的通道注册到选择器对象中。一个表示通道和选择器的键将会被返回。选择键会记住您关心的通道。它们也会追踪对应的通道是否已经就绪。当您调用一个选择器对象的 *select( )*方法时，相关的键建会被更新，用来检查所有被注册到该选择器的通道。您可以获取一个键的集合，从而找到当时已经就绪的通道。通过遍历这些键，您可以选择出每个从上次您调用 *select( )*开始直到现在，已经就绪的通道。

这是在 3000 英尺高的地方看到的情景。现在，让我们看看在地面上（甚至地下）到底发生了什么。

现在，您可能已经想要跳到例 4-1，并快速地浏览一下代码了。通过在这里和那段代码之间的内容，您将学到这些新类是如何工作的。在掌握了前面的段落里的高层次的信息之后，您需要了解选择器模型是如何在实践中被使用的。

从最基础的层面来看，选择器提供了询问通道是否已经准备好执行每个 I/0 操作的能力。例如，我们需要了解一个 `SocketChannel` 对象是否还有更多的字节需要读取，或者我们需要知道 `ServerSocketChannel` 是否有需要准备接受的连接。

在与 SelectableChannel 联合使用时，选择器提供了这种服务，但这里面有更多的事情需要去了解。就绪选择的真正价值在于潜在的大量的通道可以同时进行就绪状态的检查。调用者可以轻松地决定多个通道中的哪一个准备好要运行。有两种方式可以选择：被激发的线程可以处于休眠状态，直到一个或者多个注册到选择器的通道就绪，或者它也可以周期性地轮询选择器，看看从上次检查之后，是否有通道处于就绪状态。如果您考虑一下需要管理大量并发的连接的网络服务器(web server)的实现，就可以很容易地想到如何善加利用这些能力。

乍一看，好像只要非阻塞模式就可以模拟就绪检查功能，但实际上还不够。非阻塞模式同时还会执行您请求的任务，或指出它无法执行这项任务。这与检查它是否能够执行某种类型的操作是不同的。举个例子，如果您试图执行非阻塞操作，并且也执行成功了，您将不仅仅发现 read( )是可以执行的，同时您也已经读入了一些数据。就下来您就需要处理这些数据了。

效率上的要求使得您不能将检查就绪的代码和处理数据的代码分离开来，至少这么做会很复杂。

即使简单地询问每个通道是否已经就绪的方法是可行的，在您的代码或一个类库的包里的某些

代码需要遍历每一个候选的通道并按顺序进行检查的时候，仍然是有问题的。这会使得在检查每个通道是否就绪时都至少进行一次系统调用，这种代价是十分昂贵的，但是主要的问题是，这种检查不是原子性的。列表中的一个通道都有可能在它被检查之后就绪，但直到下一次轮询为止，您并不会觉察到这种情况。最糟糕的是，您除了不断地遍历列表之外将别无选择。您无法在某个您感兴趣的通道就绪时得到通知。

这就是为什么传统的监控多个 socket 的 Java 解决方案是为每个 socket 创建一个线程并使得线程可以在 *read( )*调用中阻塞，直到数据可用。这事实上将每个被阻塞的线程当作了 socket 监控器，并将 Java 虚拟机的线程调度当作了通知机制。这两者本来都不是为了这种目的而设计的。程序员和 Java 虚拟机都为管理所有这些线程的复杂性和性能损耗付出了代价，这在线程数量的增长失控时表现得更为突出。

真正的就绪选择必须由操作系统来做。操作系统的一项最重要的功能就是处理 I/O 请求并通知各个线程它们的数据已经准备好了。选择器类提供了这种抽象，使得 Java 代码能够以可移植的方式，请求底层的操作系统提供就绪选择服务。

让我们看一下 `java.nio.channels` 包中处理就绪选择的特定的类。

4.1.1 选择器，可选择通道和选择键类

现在，您也许还对这些用于就绪选择的 Java 成员感到困惑。让我们来区分这些活动的零件并了解它们是如何交互的吧。图 4-1 的 UML 图使得情形看起来比真实的情况更为复杂了。看看图 4-2，然后您会发现实际上只有三个有关的类 API，用于执行就绪选择：

选择器(Selector)

选择器类管理着一个被注册的通道集合的信息和它们的就绪状态。通道是和选择器一起被注册的，并且使用选择器来更新通道的就绪状态。当这么做的时候，可以选择将被激发的线程挂起，直到有就绪的的通道。

可选择通道(SelectableChannel)

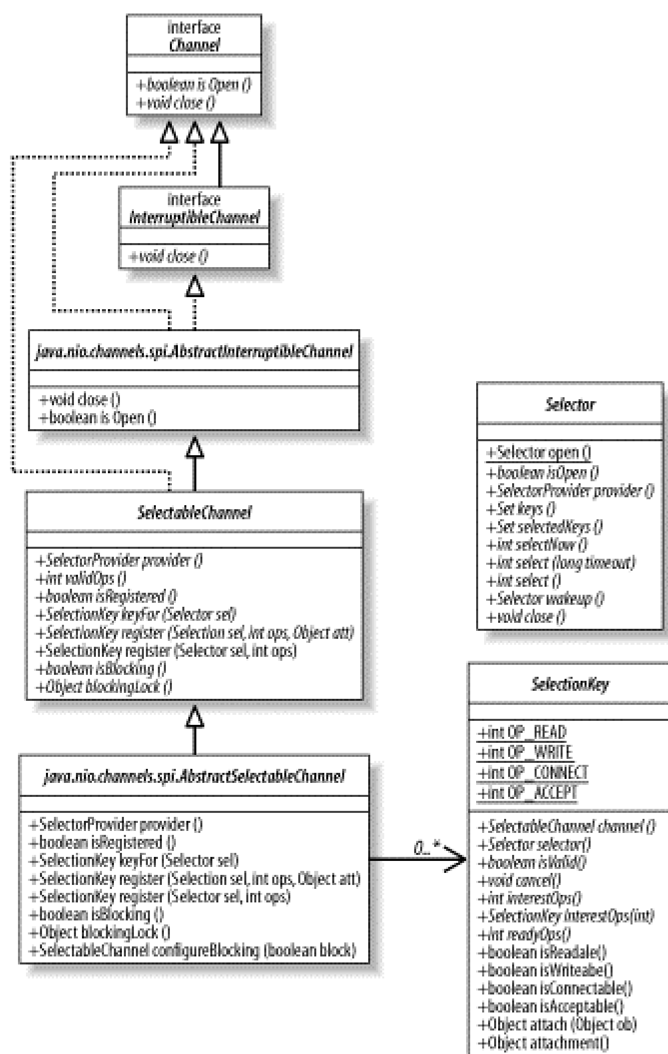
这个抽象类提供了实现通道的可选择性所需要的公共方法。它是所有支持就绪检查的通道类的父类。`FileChannel` 对象不是可选择的，因为它们没有继承 `SelectableChannel`(见图 4-2)。所有 socket 通道都是可选择的，包括从管道(Pipe)对

象的中获得的通道。`SelectableChannel` 可以被注册到 Selector 对象上，同时可以指定对那个选择器而言，那种操作是感兴趣的。一个通道可以被注册到多个选择器上，但对每个选择器而言只能被注册一次。

选择键(SelectionKey)

选 择 键 封 装 了 特 定 的 通 道 与 特 定 的 选 择 器 的 注 册 关 系 。 选 择 键 对 象 被 SelectableChannel.register( ) 返回并提供一个表示这种注册关系的标记。选择键包含了两个比特集（以整数的形式进行编码），指示了该注册关系所关心的通道操作，以及通道已经准备好的操作。

图 4-1. 就绪选择相关类的继承关系图



让我们看看 SelectableChannel 的相关 API 方法

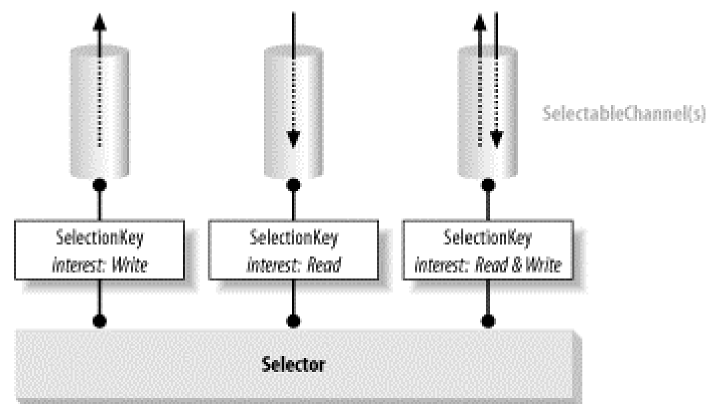
```
public abstract class SelectableChannel
        extends AbstractChannel
        implements Channel
{
        // This is a partial API listing
        public abstract SelectionKey register (Selector sel, int ops)
        throws ClosedChannelException;
        public abstract SelectionKey register (Selector sel, int ops,
        Object att)
        throws ClosedChannelException;
        public abstract boolean isRegistered( );
```

```
        public abstract SelectionKey keyFor (Selector sel);
        public abstract int validOps( );
        public abstract void configureBlocking (boolean block)
        throws IOException;
        public abstract boolean isBlocking( );
        public abstract Object blockingLock( );
}
```

非阻塞特性与多元执行特性的关系是十分密切的——以至于 `java.nio` 的架构将两者的 API 放到了一个类中。

我们已经探讨了如何用上面列出的 `SelecableChannel` 的最后三个方法来配置并检查通道的阻塞模式（详细的探讨请参考 3.5.1 小节）。通道在被注册到一个选择器上之前，必须先设置为非阻塞模式（通过调用 `configureBlocking(false)`）。

图 4-2. 就绪选择相关类的关系



调用可选择通道的 *register( )*方法会将它注册到一个选择器上。如果您试图注册一个处于阻塞状态的通道，*register( )*将抛出未检查的 `IllegalBlockingModeException` 异常。此外，通道一旦被注册，就不能回到阻塞状态。试图这么做的话，将在调用 *configureBlocking( )*方法时将抛出 `IllegalBlockingModeException` 异常。

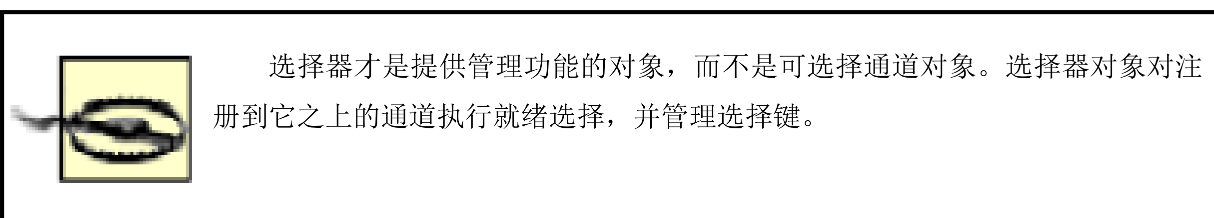
并且，理所当然地，试图注册一个已经关闭的 `SelectableChannel` 实例的话，也将抛出 `ClosedChannelException` 异常，就像方法原型指示的那样。

在我们进一步了解 *register( )*和 `SelectableChannel` 的其他方法之前，让我们先了解一下 `Selector` 类的 API，以确保我们可以更好地理解这种关系：

```
public abstract class Selector
{
        public static Selector open( ) throws IOException
        public abstract boolean isOpen( );
        public abstract void close( ) throws IOException;
        public abstract SelectionProvider provider( );
        public abstract int select( ) throws IOException;
        public abstract int select (long timeout) throws IOException;
        public abstract int selectNow( ) throws IOException;
        public abstract void wakeup( );
        public abstract Set keys( );
        public abstract Set selectedKeys( );
}
```

尽管 `SelectableChannel` 类上定义了 *register( )*方法，还是应该将通道注册到选择器上，而不是另一种方式。选择器维护了一个需要监控的通道的集合。一个给定的通道可以被注册到多于一个的选择器上，而且不需要知道它被注册了那个 Selector 对象上。将 *register( )*放在 `SelectableChannel` 上而不是 `Selector` 上，这种做法看起来有点随意。它将返回一个封装了两个对象的关系的选择键对象。重要的是要记住选择器对象控制了被注册到它之上的通道的选择过程。

```
public abstract class SelectionKey
{
        public static final int OP_READ
        public static final int OP_WRITE
        public static final int OP_CONNECT
        public static final int OP_ACCEPT
        public abstract SelectableChannel channel( );
        public abstract Selector selector( );
        public abstract void cancel( );
        public abstract boolean isValid( );
        public abstract int interestOps( );
        public abstract void interestOps (int ops);
        public abstract int readyOps( );
        public final boolean isReadable( )
        public final boolean isWritable( )
        public final boolean isConnectable( )
        public final boolean isAcceptable( )
        public final Object attach (Object ob)
        public final Object attachment( )
}
```



对于键的 interest（感兴趣的操作）集合和 ready（已经准备好的操作）集合的解释是和特定的通道相关的。每个通道的实现，将定义它自己的选择键类。在 register( )方法中构造它并将它传递给所提供的选择器对象。

在下面的章节里，我们将了解关于这三个类的方法的更多细节。

4.1.2 建立选择器

现在您可能仍然感到困惑，您在前面的三个清单中看到了大量的方法，但无法分辨出它们具体做什么，或者它们代表了什么意思。在钻研所有这一切的细节之前，让我们看看一个经典的应用实例。它可以帮助我们将所有东西放到一个特定的上下文中去理解。

为了建立监控三个 Socket 通道的选择器，您需要做像这样的事情（参见图 4-2）：

```
Selector selector = Selector.open( );
channel1.register (selector, SelectionKey.OP_READ);
```

```
channel2.register (selector, SelectionKey.OP_WRITE);
channel3.register (selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
// Wait up to 10 seconds for a channel to become ready
readyCount = selector.select (10000);
```

这些代码创建了一个新的选择器，然后将这三个(已经存在的)socket 通道注册到选择器上，而且感兴趣的操作各不相同。

*select( )*方法在将线程置于睡眠状态，直到这些刚兴趣的事情中的操作中的一个发生或者 10 秒钟的时间过去。

现在让我们看看 Selector 的 API 的细节：

```
public abstract class Selector
{
        // This is a partial API listing
        public static Selector open( ) throws IOException
        public abstract boolean isOpen( );
        public abstract void close( ) throws IOException;
        public abstract SelectionProvider provider( );
}
```

`Selector` 对象是通过调用静态工厂方法 *open( )*来实例化的。选择器不是像通道或流(stream)那样的基本 I/O 对象：数据从来没有通过它们进行传递。类方法 open( )向 SPI 发出请求，通过默认的 `SelectorProvider` 对象获取一个新的实例。通过调用一个自定义的 `SelectorProvider` 对象的 *openSelector( )*方法来创建一个 `Selector` 实例也是可行的。您可以通过调用 *provider( )*方法来决定由哪个 SelectorProvider 对象来创建给定的 `Selector` 实例。大多数情况下，您不需要关心 SPI;只需要调用 *open( )*方法来创建新的 `Selector` 对象。在那些您必须处理它们的罕见的情况下，您可以参考在附录 B 中总结的通道的 SPI 包。

继续关于将 Select 作为 I/O 对象进行处理的话题的探讨：当您不再使用它时，需要调用 *close( )*方法来释放它可能占用的资源并将所有相关的选择键设置为无效。一旦一个选择器被关闭，试图调用它的大多数方法都将导致 `ClosedSelectorException`。注意 `ClosedSelectorException` 是一个非检查(运行时的)错误。您可以通过 isOpen( )方法来测试一个选择器是否处于被打开的状态。

我们将结束对 `Selector` 的 API 的探讨，但现在先让我们看看如何将通道注册到选择器上。下面是一个之前章节中出现过的 `SelectableChannel` 的 API 的简化版本：

```
public abstract class SelectableChannel
        extends AbstractChannel
        implements Channel
{
        // This is a partial API listing
        public abstract SelectionKey register (Selector sel, int ops)
        throws ClosedChannelException;
        public abstract SelectionKey register (Selector sel, int ops,
        Object att)
        throws ClosedChannelException;
        public abstract boolean isRegistered( );
        public abstract SelectionKey keyFor (Selector sel);
        public abstract int validOps( );
}
```

就像之前提到的那样，*register( )*方法位于 `SelectableChannel` 类，尽管通道实际上是被注

册到选择器上的。您可以看到 *register( )*方法接受一个 Selector 对象作为参数，以及一个名为 ops 的整数参数。第二个参数表示所关心的通道操作。这是一个表示选择器在检查通道就绪状态时需要关心的操作的比特掩码。特定的操作比特值在 SelectonKey 类中被定义为 public static 字段。

在 JDK 1.4 中，有四种被定义的可选择操作：读(read)，写(write)，连接(connect)和接受(accept)。

并非所有的操作都在所有的可选择通道上被支持。例如，SocketChannel 不支持 accept。试图注册不支持的操作将导致 IllegalArgumentException。您可以通过调用 *validOps( )*方法来获取特定的通道所支持的操作集合。我们可以在第三章中探讨的 socket 通道类中看到这些方法。

选择器包含了注册到它们之上的通道的集合。在任意给定的时间里，对于一个给定的选择器和一个给定的通道而言，只有一种注册关系是有效的。但是，将一个通道注册到多于一个的选择器上允许的。这么做的话，在更新 interest 集合为指定的值的同时，将返回与之前相同的选择键。实际上，后续的注册都只是简单地将与之前的注册关系相关的键进行更新（见 4.2 小节）。

一个例外的情形是当您试图将一个通道注册到一个相关的键已经被取消的选择器上，而通道仍然处于被注册的状态的时候。通道不会在键被取消的时候立即注销。直到下一次操作发生为止，它们仍然会处于被注册的状态(见 4.3 小节)。在这种情况下，未检查的 CancelledKeyException 将会被抛出。请务必在键可能被取消的情况下检查 SelectionKey 对象的状态。

在之前的清单中，您可能已经注意到了 *register( )*的第二个版本，这个版本接受 object 参数。这是一个方便的方法，可以传递您提供的对象引用，在调用新生成的选择键的 *attach( )*方法时会将这个对象引用返回给您。我们将会在下一节更进一步地了解 SelectionKey 的 API。

一个单独的通道对象可以被注册到多个选择器上。可以调用 *isRegistered( )*方法来检查一个通道是否被注册到任何一个选择器上。这个方法没有提供关于通道被注册到哪个选择器上的信息，而只能知道它至少被注册到了一个选择器上。此外，在一个键被取消之后，直到通道被注销为止，可能有时间上的延迟。这个方法只是一个提示，而不是确切的答案。

任何一个通道和选择器的注册关系都被封装在一个 SelectionKey 对象中。*keyFor( )*方法将返回与该通道和指定的选择器相关的键。如果通道被注册到指定的选择器上，那么相关的键将被返回。如果它们之间没有注册关系，那么将返回 null。

4.2 使用选择键

让我们看看 SelectionKey 类的 API：

```
package java.nio.channels;
public abstract class SelectionKey
{
        public static final int OP_READ
```

```
        public static final int OP_WRITE
        public static final int OP_CONNECT
        public static final int OP_ACCEPT
        public abstract SelectableChannel channel( );
        public abstract Selector selector( );
        public abstract void cancel( );
        public abstract boolean isValid( );
        public abstract int interestOps( );
        public abstract void interestOps (int ops);
        public abstract int readyOps( );
        public final boolean isReadable( )
        public final boolean isWritable( )
        public final boolean isConnectable( )
        public final boolean isAcceptable( )
        public final Object attach (Object ob)
        public final Object attachment( )
}
```

就像之前提到的那样，一个键表示了一个特定的通道对象和一个特定的选择器对象之间的注册关系。您可以看到前两个方法中反映了这种关系。*channel( )*方法返回与该键相关的 `SelectableChannel` 对象，而 *selector( )*则返回相关的 Selector 对象。这没有什么令人惊奇的。

键对象表示了一种特定的注册关系。当应该终结这种关系的时候，可以调用 `SelectionKey` 对象的 *cancel( )*方法。可以通过调用 *isValid( )*方法来检查它是否仍然表示一种有效的关系。当键被取消时，它将被放在相关的选择器的已取消的键的集合里。注册不会立即被取消，但键会立即失效（参见 4.3 节）。当再次调用 *select( )*方法时（或者一个正在进行的 select()调用结束时），已取消的键的集合中的被取消的键将被清理掉，并且相应的注销也将完成。通道会被注销，而新的 `SelectionKey` 将被返回。

当通道关闭时，所有相关的键会自动取消（记住，一个通道可以被注册到多个选择器上）。当选择器关闭时，所有被注册到该选择器的通道都将被注销，并且相关的键将立即被无效化（取消）。一旦键被无效化，调用它的与选择相关的方法就将抛出 `CancelledKeyException`。

一个 `SelectionKey` 对象包含两个以整数形式进行编码的比特掩码：一个用于指示那些通道/选择器组合体所关心的操作(instrest 集合)，另一个表示通道准备好要执行的操作（ready 集合）。当前的 interest 集合可以通过调用键对象的 interestOps( )方法来获取。最初，这应该是通道被注册时传进来的值。这个 interset 集合永远不会被选择器改变，但您可以通过调用 *interestOps( )*方法并传入一个新的比特掩码参数来改变它。interest 集合也可以通过将通道注册到选择器上来改变（实际上使用一种迂回的方式调用 *interestOps( )*），就像 4.1.2 小节中描的那样。当相关的 `Selector` 上的 *select( )*操作正在进行时改变键的 interest 集合，不会影响那个正在进行的选择操作。所有更改将会在 select( )的下一个调用中体现出来。

可以通过调用键的 *readyOps( )*方法来获取相关的通道的已经就绪的操作。ready 集合是 interest 集合的子集，并且表示了 interest 集合中从上次调用 *select( )*以来已经就绪的那些操作。例如，下面的代码测试了与键关联的通道是否就绪。如果就绪，就将数据读取出来，写入一个缓冲区，并将它送到一个 consumer（消费者）方法中。

```
if ((key.readyOps( ) & SelectionKey.OP_READ) != 0)
{
```

```
        myBuffer.clear( );
        key.channel( ).read (myBuffer);
        doSomethingWithBuffer (myBuffer.flip( ));
}
```

就像之前提到过的那样，有四个通道操作可以被用于测试就绪状态。您可以像上面的代码那样，通过测试比特掩码来检查这些状态，但 SelectionKey 类定义了四个便于使用的布尔方法来为您测试这些比特值：*isReadable( )*，*isWritable( )*，*isConnectable( )*， 和 *isAcceptable( )*。每一个方法都与使用特定掩码来测试 *readyOps( )*方法的结果的效果相同。例如：

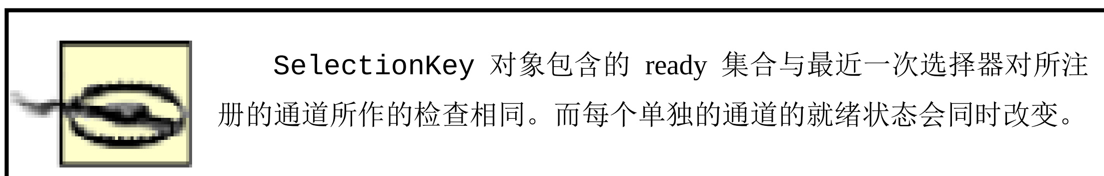
```
if (key.isWritable( ))
```

等价于：

```
if ((key.readyOps( ) & SelectionKey.OP_WRITE) != 0)
```

这四个方法在任意一个 SelectionKey 对象上都能安全地调用。不能在一个通道上注册一个它不支持的操作，这种操作也永远不会出现在 ready 集合中。调用一个不支持的操作将总是返回 false，因为这种操作在该通道上永远不会准备好。

需要注意的是，通过相关的选择键的 *readyOps( )*方法返回的就绪状态指示只是一个提示，不是保证。底层的通道在任何时候都会不断改变。其他线程可能在通道上执行操作并影响它的就绪状态。同时，操作系统的特点也总是需要考虑的。



SelectionKey 对象包含的 ready 集合与最近一次选择器对所注册的通道所作的检查相同。而每个单独的通道的就绪状态会同时改变。

您可能会从 SelectionKey 的 API 中注意到尽管有获取 ready 集合的方法，但没有重新设置那个集合的成员方法。事实上，您不能直接改变键的 ready 集合。在下一节里，也就是描述选择过程时，我们将会看到选择器和键是如何进行交互，以提供实时更新的就绪指示的。

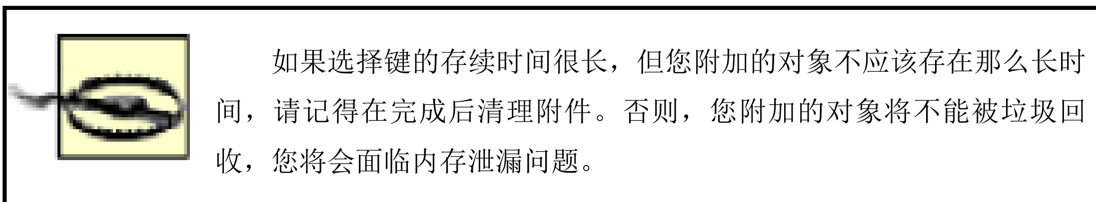
让我们试验一下 SelectionKey 的 API 中剩下的两个方法：

```
public abstract class SelectionKey
{
        // This is a partial API listing
        public final Object attach (Object ob)
        public final Object attachment( )
}
```

这两个方法允许您在键上放置一个“附件”，并在后面获取它。这是一种允许您将任意对象与键关联的便捷的方法。这个对象可以引用任何对您而言有意义的对象，例如业务对象、会话句柄、其他通道等等。这将允许您遍历与选择器相关的键，使用附加在上面的对象句柄作为引用来获取相关的上下文。

*attach( )*方法将在键对象中保存所提供的对象的引用。SelectionKey 类除了保存它之外，不会将它用于任何其他用途。任何一个之前保存在键中的附件引用都会被替换。可以使用 null 值来清除附件。可以通过调用 *attachment( )*方法来获取与键关联的附件句柄。如果没有附件，或者显式地

通过 null 方法进行过设置，这个方法将返回 null。



如果选择键的存续时间很长，但您附加的对象不应该存在那么长时间，请记得在完成后清理附件。否则，您附加的对象将不能被垃圾回收，您将会面临内存泄漏问题。

SelectableChannel 类的一个 *register( )*方法的重载版本接受一个 Object 类型的参数。这是一个方便您在注册时附加一个对象到新生成的键上的方法。以下代码：

```
SelectionKey key =
      channel.register (selector, SelectionKey.OP_READ, myObject);
```

等价于：

```
SelectionKey key = channel.register (selector, SelectionKey.OP_READ);
key.attach (myObject);
```

关于 SelectionKey 的最后一件需要注意的事情是并发性。总体上说，SelectionKey 对象是线程安全的，但知道修改 interest 集合的操作是通过 Selector 对象进行同步的是很重要的。这可能会导致 *interestOps( )*方法的调用会阻塞不确定长的一段时间。选择器所使用的锁策略（例如是否在整个选择过程中保持这些锁）是依赖于具体实现的。幸好，这种多元处理能力被特别地设计为可以使用单线程来管理多个通道。被多个线程使用的选择器也只会在系统特别复杂时产生问题。坦白地说，如果您在多线程中共享选择器时遇到了同步的问题，也许您需要重新思考一下您的设计。

我们已经探讨了 SelectionKey 的 API，但我们还没有谈完选择键的一切——远远没有。让我们进一步了解如何使用选择器管理键吧。

4.3 使用选择器

既然我们已经很好地掌握了了各种不同类以及它们之间的关联，那么现在让我们进一步了解 Selector 类，也就是就绪选择的核心。这里是 Selector 类的可用的 API。在 4.1.2 小节中，我们已经看到如何创建新的选择器，那么那些方法还剩下：

4.3.1 选择过程

在详细了解 API 之前，您需要知道一点和 Selector 内部工作原理相关的知识。就像上面探讨的那样，选择器维护着注册过的通道的集合，并且这些注册关系中的任意一个都是封装在 SelectionKey 对象中的。每一个 Selector 对象维护三个键的集合：

```
public abstract class Selector
{
        // This is a partial API listing
        public abstract Set keys( );
        public abstract Set selectedKeys( );
```

```
        public abstract int select( ) throws IOException;
        public abstract int select (long timeout) throws IOException;
        public abstract int selectNow( ) throws IOException;
        public abstract void wakeup( );
}
```

已注册的键的集合（Registered key set）

与选择器关联的已经注册的键的集合。并不是所有注册过的键都仍然有效。这个集合通过 *keys( )*方法返回，并且可能是空的。这个已注册的键的集合不是可以直接修改的；试图这么做的话将引 java.lang.UnsupportedOperationException。

已选择的键的集合（Selected key set）

已注册的键的集合的子集。这个集合的每个成员都是相关的通道被选择器（在前一个选择操作中）判断为已经准备好的，并且包含于键的 interest 集合中的操作。这个集合通过 *selectedKeys( )*方法返回（并有可能是空的）。

不要将已选择的键的集合与 ready 集合弄混了。这是一个键的集合，每个键都关联一个已经准备好至少一种操作的通道。每个键都有一个内嵌的 ready 集合，指示了所关联的通道已经准备好的操作。

键可以直接从这个集合中移除，但不能添加。试图向已选择的键的集合中添加元素将抛出 java.lang.UnsupportedOperationException。

已取消的键的集合（Cancelled key set）

已注册的键的集合的子集，这个集合包含了 *cancel( )*方法被调用过的键（这个键已经被无效化），但它们还没有被注销
。这个集合是选择器对象的私有成员，因而无法直接访问。

在一个刚初始化的 Selector 对象中，这三个集合都是空的。

Selector 类的核心是选择过程。这个名词您已经在之前看过多次了——现在应该解释一下了。基本上来说，选择器是对 *select( )*、*poll( )*等本地调用(native call)或者类似的操作系统特定的系统调用的一个包装。但是 Selector 所作的不仅仅是简单地向本地代码传送参数。它对每个选择操作应用了特定的过程。对这个过程的理解是合理地管理键和它们所表示的状态信息的基础。

选择操作是当三种形式的 select( )中的任意一种被调用时，由选择器执行的。不管是哪一种形式的调用，下面步骤将被执行：

1.已取消的键的集合将会被检查。如果它是非空的，每个已取消的键的集合中的键将从另外两个集合中移除，并且相关的通道将被注销。这个步骤结束后，已取消的键的集合将是空的。

2.已注册的键的集合中的键的 interest 集合将被检查。在这个步骤中的检查执行过后，对 interest 集合的改动不会影响剩余的检查过程。

一旦就绪条件被定下来，底层操作系统将会进行查询，以确定每个通道所关心的操作的真实就绪状态。依赖于特定的 *select( )*方法调用，如果没有通道已经准备好，线程可能会在这时阻塞，通常会有一个超时值。

直到系统调用完成为止，这个过程可能会使得调用线程睡眠一段时间，然后当前每个通道的就绪状态将确定下来。对于那些还没准备好的通道将不会执行任何的操作。对于那些操作系统指示至少已经准备好 interest 集合中的一种操作的通道，将执行以下两种操作中的一种：

a.如果通道的键还没有处于已选择的键的集合中，那么键的 ready 集合将被清空，然后表示操作系统发现的当前通道已经准备好的操作的比特掩码将被设置。

b.否则，也就是键在已选择的键的集合中。键的 ready 集合将被表示操作系统发现的当前已经准备好的操作的比特掩码更新。所有之前的已经不再是就绪状态的操作不会被清除。事实上，所有的比特位都不会被清理。由操作系统决定的 ready 集合是与之前的 ready 集合按位分离的，一旦键被放置于选择器的已选择的键的集合中，它的 ready 集合将是累积的。比特位只会被设置，不会被清理。

3.步骤 2 可能会花费很长时间，特别是所激发的线程处于休眠状态时。与该选择器相关的键可能会同时被取消。当步骤 2 结束时，步骤 1 将重新执行，以完成任意一个在选择进行的过程中，键已经被取消的通道的注销。

4.select 操作返回的值是 ready 集合在步骤 2 中被修改的键的数量，而不是已选择的键的集合中的通道的总数。返回值不是已准备好的通道的总数，而是从上一个 *select( )*调用之后进入就绪状态的通道的数量。之前的调用中就绪的，并且在本次调用中仍然就绪的通道不会被计入，而那些在前一次调用中已经就绪但已经不再处于就绪状态的通道也不会被计入。这些通道可能仍然在已选择的键的集合中，但不会被计入返回值中。返回值可能是 0。

使用内部的已取消的键的集合来延迟注销，是一种防止线程在取消键时阻塞，并防止与正在进行的选择操作冲突的优化。注销通道是一个潜在的代价很高的操作，这可能需要重新分配资源（请记住，键是与通道相关的，并且可能与它们相关的通道对象之间有复杂的交互）。清理已取消的键，并在选择操作之前和之后立即注销通道，可以消除它们可能正好在选择的过程中执行的潜在棘手问题。这是另一个兼顾健壮性的折中方案。

`Selector` 类的 *select( )*方法有以下三种不同的形式：

这三种 select 的形式，仅仅在它们在所注册的通道当前都没有就绪时，是否阻塞的方面有所不同。最简单的没有参数的形式可以用如下方式调用：

这种调用在没有通道就绪时将无限阻塞。一旦至少有一个已注册的通道就绪，选择器的选择键就会被更新，并且每个就绪的通道的 ready 集合也将被更新。返回值将会是已经确定就绪的通道的数目。正常情况下，这些方法将返回一个非零的值，因为直到一个通道就绪前它都会阻塞。但是它也可以返回非 0 值，如果选择器的 wakeup( )方法被其他线程调用。

有时您会想要限制线程等待通道就绪的时间。这种情况下，可以使用一个接受一个超时参数的 *select( )*方法的重载形式：

这种调用与之前的例子完全相同，除了如果在您提供的超时时间（以毫秒计算）内没有通道就绪时，它将返回 0。如果一个或者多个通道在时间限制终止前就绪，键的状态将会被更新，并且方法会在那时立即返回。将超时参数指定为 0 表示将无限期等待，那么它就在各个方面都等同于使用无参数版本的 *select( )*了。

就绪选择的第三种也是最后一种形式是完全非阻塞的：

```
int n = selector.selectNow( );
```

*selectNow()*方法执行就绪检查过程，但不阻塞。如果当前没有通道就绪，它将立即返回 0。

4.3.2 停止选择过程

`Selector` 的 API 中的最后一个方法，*wakeup( )*，提供了使线程从被阻塞的 *select( )*方法中优雅地退出的能力：

```
public abstract class Selector
{
        // This is a partial API listing
        public abstract void wakeup( );
}
```

有三种方式可以唤醒在 select( )方法中睡眠的线程：

调用 wakeup( )

调用 `Selector` 对象的 *wakeup( )*方法将使得选择器上的第一个还没有返回的选择操作立即返回。如果当前没有在进行中的选择，那么下一次对 *select( )*方法的一种形式的调用将立即返回。后续的选择操作将正常进行。在选择操作之间多次调用 *wakeup( )*方法与调用它一次没有什么不同。

有时这种延迟的唤醒行为并不是您想要的。您可能只想唤醒一个睡眠中的线程，而使得后续的选择继续正常地进行。您可以通过在调用 *wakeup( )*方法后调用 *selectNow( )*方法来绕过这个问题。尽管如此，如果您将您的代码构造为合理地关注于返回值和执行选择集合，那么即使下一个 *select( )*方法的调用在没有通道就绪时就立即返回，也应该不会有什么不同。不管怎么说，您应该为可能发生的事件做好准备。

调用 close( )

如果选择器的 *close( )*方法被调用，那么任何一个在选择操作中阻塞的线程都将被唤醒，就像 *wakeup( )*方法被调用了一样。与选择器相关的通道将被注销，而键将被取消。

调用 interrupt( )

如果睡眠中的线程的 *interrupt( )*方法被调用，它的返回状态将被设置。如果被唤醒的线程之后将试图在通道上执行 I/O 操作，通道将立即关闭，然后线程将捕捉到一个异常。这是由于在第三章中已经探讨过的通道的中断语义。使用 *wakeup( )*方法将会优雅地将一个在 *select( )*方法中睡眠的线程唤醒。如果您想让一个睡眠的线程在直接中断之后继续执行，需要执行一些步骤来清理中断状态（参见 `Thread.interrupted( )`的相关文档）。

`Selector` 对象将捕捉 `InterruptedException` 异常并调用 *wakeup( )*方法。

请注意这些方法中的任意一个都不会关闭任何一个相关的通道。中断一个选择器与中断一个通道是不一样的（参见 3.3 节）。选择器不会改变任意一个相关的通道，它只会检查它们的状态。当一个在 *select( )*方法中睡眠的线程中断时，对于通道的状态而言，是不会产生歧义的。

4.3.3 管理选择键

既然我们已经理解了问题的各个部分是怎样结合在一起的，那么是时候看看它们在正常的使用中是如何交互的了。为了有效地利用选择器和键提供的信息，合理地管理键是非常重要的。

选择是累积的。一旦一个选择器将一个键添加到它的已选择的键的集合中，它就不会移除这个键。并且，一旦一个键处于已选择的键的集合中，这个键的 ready 集合将只会被设置，而不会被清理。乍一看，这好像会引起麻烦，因为选择操作可能无法表现出已注册的通道的正确状态。它提供了极大的灵活性，但把合理地管理键以确保它们表示的状态信息不会变得陈旧的任务交给了程序员。

合理地使用选择器的秘诀是理解选择器维护的选择键集合所扮演的角色。（参见 4.3.1 小节，特别是选择过程的第二步。）最重要的部分是当键已经不再在已选择的键的集合中时将会发生什么。当通道上的至少一个感兴趣的操作就绪时，键的 ready 集合就会被清空，并且当前已经就绪的操作将会被添加到 ready 集合中。该键之后将被添加到已选择的键的集合中。

清理一个 `SelectKey` 的 ready 集合的方式是将这个键从已选择的键的集合中移除。选择键的就绪状态只有在选择器对象在选择操作过程中才会修改。处理思想是只有在已选择的键的集合中的键才被认为是包含了合法的就绪信息的。这些信息将在键中长久地存在，直到键从已选择的键的集合中移除，以通知选择器您已经看到并对它进行了处理。如果下一次通道的一些感兴趣的操作发生

时，键将被重新设置以反映当时通道的状态并再次被添加到已选择的键的集合中。

这种框架提供了很多灵活性。通常的做法是在选择器上调用一次 select 操作(这将更新已选择的键的集合)，然后遍历 *selectKeys( )*方法返回的键的集合。在按顺序进行检查每个键的过程中，相关的通道也根据键的就绪集合进行处理。然后键将从已选择的键的集合中被移除（通过在 `Iterator` 对象上调用 *remove( )*方法），然后检查下一个键。完成后，通过再次调用 *select( )*方法重复这个循环。例 4-1 中的代码是典型的服务器的例子。

例 4-1. 使用 select( )来为多个通道提供服务

```
package com.ronsoft.books.nio.channels;

import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.nio.channels.SelectableChannel;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetSocketAddress;
import java.util.Iterator;

/**
 * Simple echo-back server which listens for incoming stream connections and
 * echoes back whatever it reads. A single Selector object is used to listen to
 * the server socket (to accept new connections) and all the active socket
 * channels.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class SelectSockets {
	public static int PORT_NUMBER = 1234;

	public static void main(String[] argv) throws Exception {
		new SelectSockets().go(argv);
	}

	public void go(String[] argv) throws Exception {
		int port = PORT_NUMBER;

		if (argv.length > 0) { // Override default listen port
			port = Integer.parseInt(argv[0]);
		}

		System.out.println("Listening on port " + port);

		// Allocate an unbound server socket channel
		ServerSocketChannel serverChannel = ServerSocketChannel.open();
		// Get the associated ServerSocket to bind it with
		ServerSocket serverSocket = serverChannel.socket();
		// Create a new Selector for use below
		Selector selector = Selector.open();

		// Set the port the server channel will listen to
		serverSocket.bind(new InetSocketAddress(port));

		// Set nonblocking mode for the listening socket
		serverChannel.configureBlocking(false);

		// Register the ServerSocketChannel with the Selector
		serverChannel.register(selector, SelectionKey.OP_ACCEPT);

		while (true) {
```

```
            // This may block for a long time. Upon returning, the
            // selected set contains keys of the ready channels.
            int n = selector.select();

            if (n == 0) {
                  continue; // nothing to do
            }

            // Get an iterator over the set of selected keys
            Iterator it = selector.selectedKeys().iterator();

            // Look at each key in the selected set
            while (it.hasNext()) {
                  SelectionKey key = (SelectionKey) it.next();

                  // Is a new connection coming in?
                  if (key.isAcceptable()) {
                        ServerSocketChannel server =
                              (ServerSocketChannel) key.channel();
                        SocketChannel channel = server.accept();

                        registerChannel(selector, channel,
                              SelectionKey.OP_READ);

                        sayHello(channel);
                  }

                  // Is there data to read on this channel?
                  if (key.isReadable()) {
                        readDataFromSocket(key);
                  }

                  // Remove key from selected set; it's been handled
                  it.remove();
            }
      }
}

// ----------------------------------------------------------

/**
 * Register the given channel with the given selector for the given
 * operations of interest
 */
protected void registerChannel(Selector selector,
            SelectableChannel channel, int ops) throws Exception {
      if (channel == null) {
            return; // could happen
      }

      // Set the new channel nonblocking
      channel.configureBlocking(false);

      // Register it with the selector
      channel.register(selector, ops);
}

// ----------------------------------------------------------

// Use the same byte buffer for all channels. A single thread is
// servicing all the channels, so no danger of concurrent acccess.
private ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

/**
 * Sample data handler method for a channel with data ready to read.
 *
 * @param key
 *          A SelectionKey object associated with a channel determined by
 *          the selector to be ready for reading. If the channel returns
```

```
	 *          an EOF condition, it is closed here, which automatically
	 *          invalidates the associated key. The selector will then
	 *          de-register the channel on the next select call.
	 */
	protected void readDataFromSocket(SelectionKey key) throws Exception {
		SocketChannel socketChannel = (SocketChannel) key.channel();
		int count;

		buffer.clear(); // Empty buffer

		// Loop while data is available; channel is nonblocking
		while ((count = socketChannel.read(buffer)) > 0) {
			buffer.flip(); // Make buffer readable

			// Send the data; don't assume it goes all at once
			while (buffer.hasRemaining()) {
				socketChannel.write(buffer);
			}
			// WARNING: the above loop is evil. Because
			// it's writing back to the same nonblocking
			// channel it read the data from, this code can
			// potentially spin in a busy loop. In real life
			// you'd do something more useful than this.

			buffer.clear(); // Empty buffer
		}

		if (count < 0) {
			// Close channel on EOF, invalidates the key
			socketChannel.close();
		}
	}

	// ----------------------------------------------------------

	/**
	 * Spew a greeting to the incoming client connection.
	 *
	 * @param channel
	 *          The newly connected SocketChannel to say hello to.
	 */
	private void sayHello(SocketChannel channel) throws Exception {
		buffer.clear();
		buffer.put("Hi there!\r\n".getBytes());
		buffer.flip();

		channel.write(buffer);
	}
}
```

例 4-1 实现了一个简单的服务器。它创建了 ServerSocketChannel 和 Selector 对象，并将通道注册到选择器上。我们不在注册的键中保存服务器 socket 的引用，因为它永远不会被注销。这个无限循环在最上面先调用了 *select( )*，这可能会无限期地阻塞。当选择结束时，就遍历选择键并检查已经就绪的通道。

如果一个键指示与它相关的通道已经准备好执行一个 *accecpt( )*操作，我们就通过键获取关联的通道，并将它转换为 SeverSocketChannel 对象。我们都知道这么做是安全的，因为只有 ServerSocketChannel 支持 OP_ACCEPT 操作。我们也知道我们的代码只把对一个单一的 ServerSocketChannel 对象的 OP_ACCEPT 操作进行了注册。通过对服务器 socket 通道的引用，我们调用了它的 *accept( )*方法，来获取刚到达的 socket 的句柄。返回的对象的类型是 SocketChannel，也是一个可选择的通道类型。这时，与创建一个新线程来从新的连接中读取数

据不同，我们只是简单地将 socket 同多注册到选择器上。我们通过传入 OP_READ 标记，告诉选择器我们关心新的 socket 通道什么时候可以准备好读取数据。

如果键指示通道还没有准备好执行 *accept( )*，我们就检查它是否准备好执行 *read( )*。任何一个这么指示的 socket 通道一定是之前 `ServerSocketChannel` 创建的 `SocketChannel` 对象之一，并且被注册为只对读操作感兴趣。对于每个有数据需要读取的 socket 通道，我们调用一个公共的方法来读取并处理这个带有数据的 socket。需要注意的是这个公共方法需要准备好以非阻塞的方式处理 socket 上的不完整的数据。它需要迅速地返回，以其他带有后续输入的通道能够及时地得到处理。例 4-1 中只是简单地对数据进行响应，将数据写回 socket，传回给发送者。

在循环的底部，我们通过调用 `Iterator`（迭代器）对象的 *remove()*方法，将键从已选择的键的集合中移除。键可以直接从 *selectKeys()*返回的 Set 中移除，但同时需要用 `Iterator` 来检查集合，您需要使用迭代器的 *remove()*方法来避免破坏迭代器内部的状态。

4.3.4 并发性

选择器对象是线程安全的，但它们包含的键集合不是。通过 *keys( )*和 *selectKeys( )*返回的键的集合是 Selector 对象内部的私有的 Set 对象集合的直接引用。这些集合可能在任意时间被改变。已注册的键的集合是只读的。如果您试图修改它，那么您得到的奖品将是一个 `java.lang.UnsupportedOperationException`，但是当您在观察它们的时候，它们可能发生了改变的话，您仍然会遇到麻烦。`Iterator` 对象是快速失败的(fail-fast)：如果底层的 Set 被改变了，它们将会抛出 `java.util.ConcurrentModificationException`，因此如果您期望在多个线程间共享选择器和/或键，请对此做好准备。您可以直接修改选择键，但请注意您这么做时可能会彻底破坏另一个线程的 `Iterator`。

如果在多个线程并发地访问一个选择器的键的集合的时候存在任何问题，您可以采取一些步骤来合理地同步访问。在执行选择操作时，选择器在 `Selector` 对象上进行同步，然后是已注册的键的集合，最后是已选择的键的集合，按照这样的顺序。已取消的键的集合也在选择过程的的第 1 步和第 3 步之间保持同步（当与已取消的键的集合相关的通道被注销时）。

在多线程的场景中，如果您需要对任何一个键的集合进行更改，不管是直接更改还是其他操作带来的副作用，您都需要首先以相同的顺序，在同一对象上进行同步。锁的过程是非常重要的。如果竞争的线程没有以相同的顺序请求锁，就将会有死锁的潜在隐患。如果您可以确保否其他线程不会同时访问选择器，那么就不必要进行同步了。

`Selector` 类的 *close( )*方法与 *slect( )*方法的同步方式是一样的，因此也有一直阻塞的可能性。在选择过程还在进行的过程中，所有对 *close( )*的调用都会被阻塞，直到选择过程结束，或者执行选择的线程进入睡眠。在后面的情况下，执行选择的线程将会在执行关闭的线程获得锁是立即被唤醒，并关闭选择器（参见 4.3.2 小节）。

4.4 异步关闭能力

任何时候都有可能关闭一个通道或者取消一个选择键。除非您采取步骤进行同步，否则键的状态及相关的通道将发生意料之外的改变。一个特定的键的集合中的一个键的存在并不保证键仍然是有效的，或者它相关的通道仍然是打开的。

关闭通道的过程不应该是一个耗时的操作。NIO 的设计者们特别想要阻止这样的可能性：一个线程在关闭一个处于选择操作中的通道时，被阻塞于无限期的等待。当一个通道关闭时，它相关的键也就都被取消了。这并不会影响正在进行的 select( )，但这意味着在您调用 select( )之前仍然是有效的键，在返回时可能会变为无效。您总是可以使用由选择器的 selectKeys( )方法返回的已选择的键的集合：请不要自己维护键的集合。理解 3.4.5 小节描述的选择过程，对于避免遇到问题而言是非常重要的。

您可以参考 4.3.2 小节，以详细了解一个在 *select( )*中阻塞的线程是如何被唤醒的。

如果您试图使用一个已经失效的键，大多数方法将抛出 `CancelledKeyException`。但是，您可以安全地从从已取消的键中获取通道的句柄。如果通道已经关闭时，仍然试图使用它的话，在大多数情况下将引发 `ClosedChannelException`。

4.5 选择过程的可扩展性

我多次提到选择器可以简化用单线程同时管理多个可选择通道的实现。使用一个线程来为多个通道提供服务，通过消除管理各个线程的额外开销，可能会降低复杂性并可能大幅提升性能。但只使用一个线程来服务所有可选择的通道是否是一个好主意呢？这要看情况。

对单 CPU 的系统而言这可能是一个好主意，因为在任何情况下都只有一个线程能够运行。通过消除在线程之间进行上下文切换带来的额外开销，总吞吐量可以得到提高。但对于一个多 CPU 的系统呢？在一个有 n 个 CPU 的系统上，当一个单一的线程线性地轮流处理每一个线程时，可能有 n-1 个 cpu 处于空闲状态。

那么让不同道请求不同的服务类的办法如何？想象一下，如果一个应用程序为大量的分布式的传感器记录信息。每个传感器在服务线程遍历每个就绪的通道时需要等待数秒钟。这在响应时间不重要时是可以的。但对于高优先级的连接（如操作命令），如果只用一个线程为所有通道提供服务，将不得不在队列中等待。不同的应用程序的要求也是不同的。您采用的策略会受到您尝试解决的问题的影响。

在第一个场景中，如果您想要将更多的线程来为通道提供服务，请抵抗住使用多个选择器的欲望。在大量通道上执行就绪选择并不会有很大的开销，大多数工作是由底层操作系统完成的。管理多个选择器并随机地将通道分派给它们当中的一个并不是这个问题的合理的解决方案。这只会形成这个场景的一个更小的版本。

一个更好的策略是对所有的可选择通道使用一个选择器，并将对就绪通道的服务委托给其他线程。您只用一个线程监控通道的就绪状态并使用一个协调好的工作线程池来处理共接收到的数据。根据部署的条件，线程池的大小是可以调整的（或者它自己进行动态的调整）。对可选择通道的管理仍然是简单的，而简单的就是好的。

第二个场景中，某些通道要求比其他通道更高的响应速度，可以通过使用两个选择器来解决：一个为命令连接服务，另一个为普通连接服务。但这种场景也可以使用与第一个场景十分相似的办法来解决。与将所有准备好的通道放到同一个线程池的做法不同，通道可以根据功能由不同的工作线程来处理。它们可能可以是日志线程池，命令/控制线程池，状态请求线程池，等等。

例 4-2 的代码是例 4-1 的一般性的选择循环的扩展。它覆写了 *readDataFromSocket( )*方法，并使用线程池来为准备好数据用于读取的通道提供服务。与在主线程中同步地读取数据不同，这个版本的实现将 `SelectionKey` 对象传递给为其服务的工作线程。

例 4-2. 使用线程池来为通道提供服务

```
package com.ronsoft.books.nio.channels;

import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.channels.SelectionKey;
import java.util.List;
import java.util.LinkedList;
import java.io.IOException;

/**
 * Specialization of the SelectSockets class which uses a thread pool to service
 * channels. The thread pool is an ad-hoc implementation quicky lashed togther
 * in a few hours for demonstration purposes. It's definitely not production
 * quality.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class SelectSocketsThreadPool extends SelectSockets {
    private static final int MAX_THREADS = 5;
    private ThreadPool pool = new ThreadPool(MAX_THREADS);

    // -------------------------------------------------------------

    public static void main(String[] argv) throws Exception {
          new SelectSocketsThreadPool().go(argv);
    }

    // -------------------------------------------------------------

    /**
     * Sample data handler method for a channel with data ready to read. This
     * method is invoked from the go( ) method in the parent class. This handler
     * delegates to a worker thread in a thread pool to service the channel,
     * then returns immediately.
     *
     * @param key
     *          A SelectionKey object representing a channel determined by the
     *          selector to be ready for reading. If the channel returns an
     *          EOF condition, it is closed here, which automatically
     *          invalidates the associated key. The selector will then
     *          de-register the channel on the next select call.
     */
    protected void readDataFromSocket(SelectionKey key) throws Exception {
```

```
		WorkerThread worker = pool.getWorker();

		if (worker == null) {
			// No threads available. Do nothing. The selection
			// loop will keep calling this method until a
			// thread becomes available. This design could
			// be improved.
			return;
		}

		// Invoking this wakes up the worker thread, then returns
		worker.serviceChannel(key);
	}

	// ---------------------------------------------------------------

	/**
	 * A very simple thread pool class. The pool size is set at construction
	 * time and remains fixed. Threads are cycled through a FIFO idle queue.
	 */
	private class ThreadPool {
		List idle = new LinkedList();

		ThreadPool(int poolSize) {
			// Fill up the pool with worker threads
			for (int i = 0; i < poolSize; i++) {
				WorkerThread thread = new WorkerThread(this);

				// Set thread name for debugging. Start it.
				thread.setName("Worker" + (i + 1));
				thread.start();

				idle.add(thread);
			}
		}

		/**
		 * Find an idle worker thread, if any. Could return null.
		 */
		WorkerThread getWorker() {
			WorkerThread worker = null;

			synchronized (idle) {
				if (idle.size() > 0) {
					worker = (WorkerThread) idle.remove(0);
				}
			}

			return (worker);
		}

		/**
		 * Called by the worker thread to return itself to the idle pool.
		 */
		void returnWorker(WorkerThread worker) {
			synchronized (idle) {
				idle.add(worker);
			}
		}
	}

	/**
	 * A worker thread class which can drain channels and echo-back the input.
	 * Each instance is constructed with a reference to the owning thread pool
	 * object. When started, the thread loops forever waiting to be awakened to
	 * service the channel associated with a SelectionKey object. The worker is
	 * tasked by calling its serviceChannel( ) method with a SelectionKey
	 * object. The serviceChannel( ) method stores the key reference in the
	 * thread object then calls notify( ) to wake it up. When the channel has
```

```
 * been drained, the worker thread returns itself to its parent pool.
 */
private class WorkerThread extends Thread {
	private ByteBuffer buffer = ByteBuffer.allocate(1024);
	private ThreadPool pool;
	private SelectionKey key;

	WorkerThread(ThreadPool pool) {
		this.pool = pool;
	}

	// Loop forever waiting for work to do
	public synchronized void run() {
		System.out.println(this.getName() + " is ready");
		while (true) {
			try {
				// Sleep and release object lock
				this.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
				// Clear interrupt status
				this.interrupted();
			}

			if (key == null) {
				continue; // just in case
			}

			System.out.println(this.getName() + " has been awakened");
			try {
				drainChannel(key);
			} catch (Exception e) {
				System.out.println("Caught '" + e
					+ "' closing channel");

				// Close channel and nudge selector
				try {
					key.channel().close();
				} catch (IOException ex) {
					ex.printStackTrace();
				}

				key.selector().wakeup();
			}
			key = null;

			// Done. Ready for more. Return to pool
			this.pool.returnWorker(this);
		}
	}

	/**
	 * Called to initiate a unit of work by this worker thread on the
	 * provided SelectionKey object. This method is synchronized, as is the
	 * run( ) method, so only one key can be serviced at a given time.
	 * Before waking the worker thread, and before returning to the main
	* selection loop, this key's interest set is updated to remove OP_READ.
	 * This will cause the selector to ignore read-readiness for this
	 * channel while the worker thread is servicing it.
	 */
	synchronized void serviceChannel(SelectionKey key) {
		this.key = key;

		key.interestOps(key.interestOps() & (~SelectionKey.OP_READ));

		this.notify(); // Awaken the thread
	}

	/**
```

```
         * The actual code which drains the channel associated with the given
         * key. This method assumes the key has been modified prior to
         * invocation to turn off selection interest in OP_READ. When this
         * method completes it re-enables OP_READ and calls wakeup( ) on the
         * selector so the selector will resume watching this channel.
         */
        void drainChannel(SelectionKey key) throws Exception {
            SocketChannel channel = (SocketChannel) key.channel();
            int count;

            buffer.clear(); // Empty buffer

            // Loop while data is available; channel is nonblocking
            while ((count = channel.read(buffer)) > 0) {
                buffer.flip(); // make buffer readable

                // Send the data; may not go all at once
                while (buffer.hasRemaining()) {
                    channel.write(buffer);
                }
                // WARNING: the above loop is evil.
                // See comments in superclass.

                buffer.clear(); // Empty buffer
            }

            if (count < 0) {
                // Close channel on EOF; invalidates the key
                channel.close();
                return;
            }

            // Resume interest in OP_READ
            key.interestOps(key.interestOps() | SelectionKey.OP_READ);

            // Cycle the selector so this key is active again
            key.selector().wakeup();
        }
    }
}
```

由于执行选择过程的线程将重新循环并几乎立即再次调用 select( )，键的 interest 集合将被修改，并将 interest（感兴趣的操作）从读取就绪(read-rreadiness)状态中移除。这将防止选择器重复地调用 *readDataFromSocket( )*（因为通道仍然会准备好读取数据，直到工作线程从它那里读取数据）。当工作线程结束为通道提供的服务时，它将再次更新键的 ready 集合，来将 interest 重新放到读取就绪集合中。它也会在选择器上显式地嗲用 *wakeup( )*。如果主线程在 *select( )*中被阻塞，这将使它继续执行。这个选择循环会再次执行一个轮回（可能什么也没做）并带着被更新的键重新进入 *select( )*。

4.6 总结

在本章中，我们介绍了 NIO 最强大的一面。就绪选择对大规模、高容量的服务器端应用程序来说是非常必要的。将这种能力补充到 Java 平台中，意味着企业级 Java 应用程序可以和用其他编程语言编写的有可比性的应用程序一较高下了。本章中的关键概念如下：

就绪选择相关类(Selector classes)

Selector，SelectableChannel 和 SelectionKey 这三个类组成了使得在 Java 平台上进行就绪检查变得可行的三驾马车。在 4.1 小节，我们看到了这些类相互之间的关联以及它们表示的含义。

选择键(Selection keys)

在 4.2 小节中，我们学到了更多关于选择键的知识以及如何使用它们。SelectionKey 类封装了 SelectableChannel 对象和 Selector 之间的关系。

选择器(Selectors)

选择器请求操作系统决定那个注册到给定选择器上的通道已经准备好指定感兴趣的 I/O 操作。我们在 4.3 小节学习了怎样管理键集合并从 select( )的调用中返回。我们也探讨了一些与就绪选择相关的并发性问题。

异步关闭能力(Asynchronous closability)

在 4.1 小节中我们接触了关于异步关闭选择器和通道的问题。

多线程(Multithreading)

在 4.5 小节，我们探讨了怎样将多线程用于为可选择通道提供服务，而不必借助多个选择器对象来实现。

选择器为 Java 服务器应用程序提供了强有力的保证。随着这种新的强大能力整合到商业服务器应用程序中去，服务器端的应用程序将更加可扩展，更可靠，并且响应速度更快。

好了。我们已将结束了了解 java.nio 和它的子类的旅程。当请不要放下摄像机离开。我们还有更多不额外收费的奖品。重新登上公交车时请注意您的脚，下一站是正则表达式的迷人大陆。

第五章

正则表达式

Regular Expressions

嗨，这是一种魔法。

—英格兰高地人

本章我们将讨论新的程序包java.util.regex（见图 5-1）中类的API（译注 10）。JSR51，即Java规范请求（Java Specification Request），定义了新的I/O权能，它还明确了添加到Java平台上的正则表达式处理技术。尽管严格说来正则表达式并不是I/O，但是它们最常用于浏览从文件或数据流（stream）中读取的文本数据。

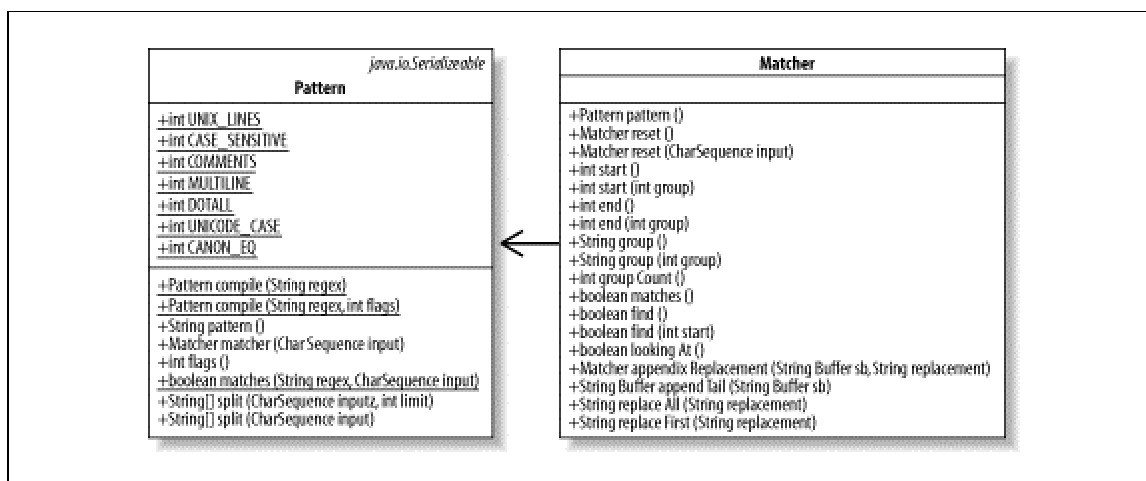


图 5-1 正则表达式类

Perl、*egrep*和其它文本处理工具有着功能强大的模式匹配（pattern matching），本章中你将学会如何使用新的Java API达到相同的模式匹配效果。关于正则表达式的详细探究不在本书的范围内，但是本书假设您熟悉正则表达式的实际作用。假若你不熟悉正则表达式，想要提升自己的技能，又或者对本章内容感到困惑，我建议你挑选一本好的参考书。O’Reilly出版了的一本正则表达式权威书籍（甚至JDK文档都引用了它）：Mastering Regular Expressions，作者Jeffrey E.F. Friedl（译注[11]）。

（http://www.oreilly.com/catalog/regex/）

10： API，Application programming interface 的缩写，应用程序接口或应用编程接口。

11： O'Reilly 是一家以出版科技图书见长的出版社，出版刊物包括一些计算机和生物方面的图书。JDK，Java Development Kit 的缩写，Java 开发工具，是目前使用最为广泛的 Java 软件开发工具包。Mastering Regular Expressions，在电子工业出版社出版，余晟译版中译为《精通正则表达式》。

5.1 正则表达式基础 Regular Expression Basics

正则表达式（regular expression）是描述或表达在目标字符序列内匹配一定字符模式的字符序列（译注 [12]）。正则表达式在Unix世界已存在多年并凭借*sed*、*grep*（查找正则表达式）、*awk*等文本编辑工具而广为使用。在Unix平台上使用的正则表达式因其长久的历史已经成了大部分正则表达式处理器的基础。开放组（The Open Group）是Unix标准团体，它将正则表达式句法作为Unix规范的一部分作了详述（译注 [13]）。

（http://www.opengroup.org/onlinepubs/7908799/xbd/re.html）

广受欢迎的Perl（注 1）脚本语言将正则表达式处理技术直接纳入其语言句法的麾下。随着Perl的演变及流行，它在添加更多复杂功能的同时也扩展了正则表达式句法。Perl因其能力及灵活性而几乎无所不在，并且随后也建立了正则表达式处理技术的实际标准。

java.util.regex 的类提供的正则表达式权能（capability）效仿了 Perl 5 提供的权能。细微差别主要囿于一般用户极少接触的神秘领域。关于 java.util.regex.Pattern 类的细节，详见 5.4 节，表 5-7 及 Javadoc API 文档。

在Perl脚本中，正则表达式被作为内联的（inline，译注 [14]）子过程匹配变量值。将正则表达式赋值（evaluation）集成到语言中使得Perl在文本处理的脚本应用中广受欢迎。直到现在，Java已经难以企及。因为在文本文件中处理非平凡模式（notrivial pattern）缺乏内置工作，所以用Java处理文本文件相当累赘麻烦。

12： 本章中出现的字符序列原文为 character sequence(s)或 sequence(s) of characters，区别于下文中遇到的字符串（string）。字符串的定义：由零个或多个字符组成的有限序列。可见二者存在差异，故区别译之。

13： The Open Group 是厂商中立、技术中立的产业联盟，其设想是“信息无边界”（Boundaryless Information Flow）。Standards Group，这里译为标准团体，是根据章程制订、批准标准的团体。为了与 grammar（语法）区别，原文中 syntax 在本章中译为句法（许多地方将其翻译为“语法”）。大英袖珍百科（中文版）对二者的解释是：句法（syntax），对词在句子、分句、短语中的配列方式以及对句子构成及其各部分之间关系的研究；语法（grammar），说明音系学、形态学、句法和语义学的语言规则，亦指论述上列规则的摘要。

注 1： 实际的提取和报告语言（或称为病态折中垃圾列表器，这取决于你调试的时间有多长）。

14： inline，内联的，内嵌的。在编程中，指的是函数调用被函数主体替换，实际参数被替换为正式的参数。内联函数通常是作为编译时提高效率的变换。

什么是正则表达式？

正则表达式（regular expression）萌芽于 Stephen Kleene 在十九世纪五十年代发明的用于描述正则集（regular set）的数学符号表示法。在 1968 年以前正则表达式仍停留在数学领域，在 1968 年这年贝尔实验室研究员、Unix 先驱 Ken Thompson 开发了基于正则表达式的检索算法，并最终集成到 Unix ed 文本编辑器中。

ed 中的 g/Regular Expression/p（Global Regular Expression Print 全局正则表达式打印）命令十分有用，并由此衍生中独立的 grep 命令。其它基于正则的命令还有：*sed*、*awk*、*lex*、*egrep* 等等。随着正则表达式使用的增多及新功能的增加，用于文本检索的句法从其数学“老祖宗”中分离出来。

久而久之，“一厨一口锅，风味各不同”，正则表达式这道“菜”有了许多“风味（flavor）”。在 1986 年首次引入了 POSIX（Portable Operating System Interface，可移植的操作系统接口）标准，该标准尝试对大量的操作系统特性进行标准化。POSIX 定义了两类正则表达式：基本正则表达式（basic regular expression, BRE）和扩展正则表达式（extended regular expression, ERE）。

译注：flavor 在本章中译为“风味”，保留了其原始意思。余晟版的 Mastering Regular Expressions 将其译为“流派”。Flavor: One of several varieties of a system, having its own details of operation.（微软计算机辞典）一个系统的若干品种之一，有其各自的操作细节。

Perl享有较高水平的正则表达式集成，正当Java从未触及这一高度之际，java.util.regex包为Java带来了福音，它可以提供与Perl同等水平的表达能力。当然，Perl与Java的使用模型是不同的：Perl是过程式脚本语言，而Java是编译的、面向对象[注2]的语言。但是Java正则表达式API易于使用，并且现在允许Java轻松实现文本处理任务，而该任务在以往都是“外包”给Perl的。

[注2]：有些人可能主张 Perl 也是面向对象的。Perl 文档本身清楚地说明了 Perl“对象”顶多就是句法错觉。如果你想对此进行辩驳，那么坦白地说你很可能看错书了。

Larry Wall 在 1987 年发布了首版 perl。Perl 十分有用，因为它将许多功能强大的部件集成在一个脚本语言中，而不仅仅是将正则表达式处理技术编入语言句法结构中。首版 Perl 采用的正则表达式代码取自 James Gosling 版的 emacs（editor macros，编辑器宏指令）。Perl 现在已经发展到了第 5 版，普遍使用已达八年。它已经成了衡量大部分正则表达式处理器的基准。

正则表达式匹配引擎（regular expression-matching engine）可分成两类：确定性有限自动机（Deterministic Finite Automaton, DFA）和非确定性有限自动机（Nondeterministic Finite Automaton, NFA）。二者的差异与如何编译表达式及如何匹配目标有关。一般情况下 DFA 速度较快，因为它会建立匹配树（matching tree），根据匹配进程"修剪"无法达到的"分枝"，预先作更多的工作。NFA 可能较慢，因为它运行的是更详尽的检索，常常需要回溯（backtrack）。DFA 速度较快，但 NFA 功能更全。例如 DFA 处理器无法捕捉子表达式（subexpression）而 NFA 可以。java.util.regex 引擎是 NFA 并且它的句法与 Perl 5 相似。

虽然正则表达式成了带 JDK 1.4 的 Java 平台的正式组成部分，但是这无碍于 java.util.regex 是第一个适用于 Java 的正则表达式包这一事实。Apache 组织有两个开放源的（open source）正则表达式包：Jakarta-ORO 和 Jakarta-Regexp，距离它们发布已经有些时日了。继承自 OROMatcher 的 Jakarta-ORO 发展了 Apache 并且自身也得到了进一步的增强。它有许多功能部件且可高度自定义（highly customizable）。Jakarta-Regexp 同样 Apache 有所助益但是范围较小。GNU 的"民间人士"提供了一个名为 gnu.regexp 的正则表达式包，它有一些像 Perl 的功能。并且 IBM 有个名为 com.ibm.regex 的商业软件包。它也有许多类似 Perl 功能，可很好地支持 Unicode。

正则表达式历史、正则处理器类型、可用的实现方式及所有你希望知道的有关正则表达式句法和使用的所有细节，请查阅 Jeffrey E.F. Friedl 的著书：Mastering Regular Expressions (O'Reilly) 。

5.2 Java正则表达式API　The Java Regular Expression API

`java.util.regex` 程序包只包含用于实现 `Java` 正则表达式处理技术的两个类，分别名为 `Pattern` 和 `Matcher`。自然而然你会想到正则表达式由模式匹配（`pattern matching`）而成.`java.lang` 还定义了一个新接口，它支持这些新的类。在研究 `Patternt` 和 `Matcher` 之前，我们先快速浏览一下 `CharSequence` 这一新概念。

另外，为方便起见 `String` 类为运行正则表达式匹配提供了一些新程序作为捷径。这些将 `5.3` 节中讨论。

5.2.1 CharSequence接口　The CharSequence Interface

正则表达式是根据字符序列进行模式匹配的。虽然 String 对象封装了字符序列，但是它们并不能够这样做的唯一对象。

JDK 1.4 定义了一个名为 *CharSequence* 的新接口，可描述特定不变的字符序列。该新接口

是一个抽象(abstraction)，它把字符序列从包含这些字符的具体实现(specific implementation)中分离出来。JDK 1.4 对“年高德勋”的 String 和 StringBuffer 类进行了改进，用于实现 *CharSequence* 接口。新的 CharBuffer 类（在第二章中介绍过）也实现了 *CharSequence*。*CharSequence* 接口也在字符集映射中投入了使用（参见第六章）。

CharSequence 定义的 API 十分简单。毕竟它没有花太多“笔墨”描述字符序列。

```
package java.lang;

public interface CharSequence
{
	int length( );
	char charAt (int index);

	public String toString( );
	CharSequence subSequence (int start, int end);
}
```

CharSequence 描述的每个字符序列通过 *length( )*方法会返回某个长度值。通过调用 *charAt( )*可以得到序列的各个字符，其中索引是期望的字符位置（desired character position）。字符位置从零到字符序列的长度之间，与我们熟悉的 *String.charAt( )*基本一样。

*toString( )*方法返回的 *String* 对象包括所描述的字符序列。这可能很有用，如打印字符序列。正如之前提过的，String 现在实现了 *CharSequence*。String 和 *CharSequence* 同为不变的，因此如果 *CharSequence* 描述一个完整的 String，那么 *CharSequence* 的 *toString( )*方法返回的是潜在的 String 对象而不是副本。如果备份对象是 StringBuffer 或 CharBuffer，系统将创建一个新的 String 保存字符序列的副本。

最后通过调用 *subSequence( )*方法会创建一个新的 *CharSequence* 描述子范围（subrange）。start 和 end 的指定方式与 *String.substring( )*的方式相同：start 必须是序列的有效索引（valid index）；end 必须比 start 大，标志的是最末字符的索引加一。换句话说，start 是起始索引（计算在内），end 是结束索引（不计算在内）。

CharSequence 接口因为没有赋值方法（mutator method）看上去似乎是不变的，但是基本的实现对象可能不是不变的。*CharSequence* 方法反映了基本对象的现状。如果状态改变，*CharSequence* 方法返回的信息同样会发生变化（见例 5-1）。如何你依赖 *CharSequence* 保持稳定且不确认基础的实现，你可以调用 *toString( )*方法，对字符序列拍个真实不变的快照。

例 5-1 CharSequence 接口实例

```
package com.ronsoft.books.nio.regex;

import java.nio.CharBuffer;

/**
 * Demonstrate behavior of java.lang.CharSequence as implemented
 * by String, StringBuffer and CharBuffer.
 *
```

```
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class CharSeq
{
   public static void main (String [] argv)
   {
      StringBuffer stringBuffer = new StringBuffer ("Hello World");
      CharBuffer charBuffer = CharBuffer.allocate (20);
      CharSequence charSequence = "Hello World";

      //直接来源于String
      printCharSequence (charSequence);

      //来源于StringBuffer
      charSequence = stringBuffer;
      printCharSequence (charSequence);

      //更改StringBuffer
      stringBuffer.setLength (0);
      stringBuffer.append ("Goodbye cruel world");
      //相同、“不变的”CharSequence产生了不同的结果
      printCharSequence (charSequence);

      //从CharBuffer中导出CharSequence
      charSequence = charBuffer;
      charBuffer.put ("xxxxxxxxxxxxxxxxxxxx");
      charBuffer.clear( );

      charBuffer.put ("Hello World");
      charBuffer.flip( );
      printCharSequence (charSequence);

      charBuffer.mark( );
      charBuffer.put ("Seeya");
      charBuffer.reset( );
      printCharSequence (charSequence);

      charBuffer.clear( );
      printCharSequence (charSequence);
      //更改基础CharBuffer会反映在只读的CharSequence接口上
   }

   private static void printCharSequence (CharSequence cs)
   {
      System.out.println ("length=" + cs.length( )
         + ", content='" + cs.toString( ) + "'");
   }
}
```

以下是执行 CharSequence 的结果：

```
length=11, content='Hello World'
length=11, content='Hello World'
length=19, content='Goodbye cruel world'
length=11, content='Hello World'
length=11, content='Seeya World'
length=20, content='Seeya Worldxxxxxxxxx'
```

5.2.2 Pattern类　The Pattern Class

Pattern 类封装了正则表达式，它是你希望在目标字符序列中检索的模式。匹配正则表达式的代价可能非常高昂，因为可能排列数量巨大，尤其是模式反复应用的情况。大部分正则

表达式处理器（包括 Perl 在内，在封装中）首先会编译表达式，然后利用编译好的表达式在输入中进行模式检测。

在这一点上 Java 正则表达式程序包别无两样。Pattern 类的实例是将一个编译好的正则表达式封装起来。让我们看看完整的 Pattern API，看看它是如何使用的。记住，这并不是一个句法完整的类文件，它只中去掉了类主体的方法签名。

```
package java.util.regex;

public final class Pattern implements java.io.Serializable
{
   public static final int UNIX_LINES
   public static final int CASE_INSENSITIVE
   public static final int COMMENTS
   public static final int MULTILINE
   public static final int DOTALL
   public static final int UNICODE_CASE
   public static final int CANON_EQ

   public static boolean matches (String regex, CharSequence input)

   public static Pattern compile (String regex)
   public static Pattern compile (String regex, int flags)

   public String pattern( )
   public int flags( )

   public String[] split (CharSequence input, int limit)
   public String[] split (CharSequence input)

   public Matcher matcher (CharSequence input)
}
```

上面所列的第一个方法 *matches( )*是个公用程序。它可以进行完整的匹配操作，并根据正则表达式是否匹配整个的（entire）输入序列返回一个布尔值。这种方法很容易上手，因为你无须追踪任何对象；你要做的仅是调用一个简单的静态方法并测试结果。

```
public boolean goodAnswer (String answer)
{
    return (Pattern.matches ("[Yy]es|[Yy]|[Tt]rue", answer));
}
```

这种方法适用于默认设置尚可接受并且只需进行一次测试的情况。假如你要重复检查同一模式，假如你要找的模式是输入的子序列，又假如你要设置非默认选项，那么你应当创建一个新的 Pattern 对象并使用新对象的 API 方法。

需要注意的是 Pattern 类并没有公用建构函数。只有通过调用静态工厂方法才可以创建新的实例。*compile( )*的两个形式采用的都是正则表达式的 String 参数。返回的 Pattern 对象包含被转换成已编译内部形式的正则表达式。如果你提供的正则表达式形态异常，那么 *compile( )*工厂方法会抛出 java.util.regex.PatternSyntaxException（模式句法异常）。这是未经检查的异常，因此如果你对自己使用的正则表达式是否可行存在疑虑（例如它传递给你是一个变量），那么你可以把对 *compile( )*的调用放到 try/catch 块中进行检测。

*compile( )*的第二种形式接受标志有一个位掩码,这影响了正则表达式的默认编译。这些标志启用了可选的编译模式行为，例如如何处理边界或不区分大小写等。（除 CANOB_EQ 外）这些标志（flag）同样可由嵌入表达式内的子表达式启用。标志可以与布尔或（OR）表达式结合使用，如下所示：

```
Pattern pattern = Pattern.compile ("[A-Z][a-zA-Z]*",
   Pattern.CASE_INSENSITIVE | Pattern.UNIX_LINES);
```

所有标志默认为关闭状态。表 5-1 总结了各个编译时间选项的意义。

表 5-1. 影响正则表达式编译的标志值

标志名	嵌入的表达式	描述
UNIX 命令行 UNIX lines	(?d)	启用 Unix 命令行模式。 在此模式中，只认定换行符（\n）是行终止符。这就影响了脱字符^和美元符号$的行为。如果未设置该标志（保持默认状态），那么如下表示皆可认为是行终止符：\n、\r、\r\n、\u0085 (下一行)、\u2028 (行分隔符)和\u2029 (段落分隔符)。 Unix 命令行模式也可由嵌入的表达式（?d）指定。
不区分大小写 CASE INSENSITIVE	(?i)	启用不区分大小写的模式匹配，这可能会使性能有点打折。 使用这个标志的前提是设置匹配的字符只来自 US-ASCII 字符集。如果你正在处理的是其它语言的字符集，你同样可以规定 UNICODE_CASE 标志，从而启用 Unicode 识别大小写折叠（Unicode-aware case folding）。
UNICODE 大小写 UNICODE CASE	(?iu)	Unicode 识别、大小写折叠模式 当它与不区分大小写标志一起使用时，不区分大小写的字符匹配与 Unicode 标准一致。这确保了平等对待所有语言中使用 Unicode 字符集编码的大写字符和小写字符。 该选项可能会降低性能。
注释 COMMENTS	(?x)	该模式允许有空白符（whitespace）和注释。 该模式生效时将忽略模式中任何空白符，并且以#字符开头的注释直到行尾都将被忽略。
多行 MULTILINE	(?m)	打开多行模式。 在多行模式中，脱字符^只匹配行终止符之后或字符序列之前的位置，美元符$只匹配行终止符之前或字符序列之后的位置。在普通模式下，这些表达式只匹配整个字符序列的开头或结尾（忽略行终止符）。
DOTALL DOTALL	(?s)	点（.）字符可匹配任意字符，甚至是行分隔符。 默认状态下，点表达式（dot expression）并不匹配行分隔符。该

		选项相当于 Perl 的单行模式，因而有（？s）这一嵌入式标志名。
规范等式 CANON EQ	无	启用规范等价模式。 当规定了这一标志，当且仅当字符的标准分解匹配时认为它们匹配。 例如，当该标志启用时，双字符序列 a\u030A（Unicode 符号“拉丁文小写字母 a”后面接一个“组合上圆圈°”，即 a°）与单字符\u00E5（拉丁文小写字母 a“头上”带圆，即 å）是匹配的。 默认状态下，并不考虑规范等价性。关于规范等价性，详情请见 http://www.unicode.org/中字符映射的定义。该标志可能会导致性能显著下降。没有嵌入式标志表达式可以启用规范等式模式。

Pattern 类的实例是不变的，各个实例与对应的正则表达式绑定，无法修改。Pattern 对象也是线程安全的，可被多个线程同时使用。

因此，一旦你拿到一个 Pattern，你能对它做什么呢？

```
package java.util.regex;
public final class Pattern implements java.io.Serializable
{
        // 这是部分的API列表
        public String pattern( )
        public int flags( )
        public String[] split (CharSequence input, int limit)
        public String[] split (CharSequence input)
        public Matcher matcher (CharSequence input)
}
```

下面是两个方法，作用是返回 Pattern 类 API 关于封装表达式的信息。*pattern( )*返回的 String 被用于初建 Pattern 实例（建立对象时字符串被传递给 *compile( )*）。另一个是 *flags( )*，返回的是在编译模式时提供的标志位掩码。如果 Pattern 对象由无参数的 *compile( )*创建，则 *flags( )*返回的值是 0。返回的值反映的只是提供给 *compile( )*的明确标志值；它不包括由正则表达式模式中的嵌入表达式设置的任何等效标志，等效标志如表 5-1 第二列所示。

实例方法 *split( )*是公用程序，它使用模式作为定界符（delimiter）来标记字符序列。这会令人联想到 StringTokenizer，但是它的功能更强，因为定界符可以是匹配正则表达式的多字符序列。另外，*split( )*方法是不监控状态的，它返回的是字符串标志的阵列而不是要求多个调用程序并循环执行它们：

```
Pattern spacePat = Pattern.compile ("\\s+");
String [] tokens = spacePat.split (input);
```

调用仅有一个参数的 *split( )*与调用两个参数但第二参数为零的该方法是等效的。*split( )*第二个参数指示了输入序列被正则表达式拆分的限制次数。限制参数的意义在于防止超负荷。非正的值有特殊的意义。

如果传递给 *split( )*的限值为负（任意负数），则字符序列将无限制地拆分直至输入穷尽。返回的阵列可能是任何长度。如果给定的限值为零，则输入将被无限拆分，但是结尾的空字符串将不在结果阵列内。如果限值为正，它设置的是返回的 String 阵列的最大值。假设限制为 n，那么正则表达式最大被用到 n-1 次。表 5-2 对这些组合进行了总结，生成该表格的代码见例 5-2。

表 5-2. split( )行为矩阵

Input: poodle zoo	Regex = " "	Regex = "d"	Regex="o"
Limit = 1	"poodle zoo"	"poodle zoo"	"poodle zoo"
Limit = 2	"poodle", "zoo"	"poo", "le zoo"	"p", "odle zoo"
Limit = 5	"poodle", "zoo"	"poo", "le ", "oo"	"p", , "dle z", ,
Limit = -2	"poodle", "zoo"	"poo", "le ", "oo"	"p", , "dle z", ,
Limit = 0	"poodle", "zoo"	"poo", "le ", "oo"	"p", , "dle z"

最后，*matcher( )*是为已编译的模式创建 Matcher 对象的工厂方法。匹配器（matcher）是监控状态的匹配引擎（stateful matching engine），它知道如何对目标字符序列的模式（Pattern 对象从何而来）进行匹配。在创建 Matcher 时你必须提供原始的输入目标，但是随后可提供不同的输入（5.2.3 节将对此进行讨论）。

5.2.2.1 利用模式类拆分字符串　Splitting strings with the Pattern class

根据若干不同的正则表达式模式和不同的限值对相同的输入字符串进行拆分，其结果如例 5-2 生成的矩阵所示。

例 5-2. 根据模式拆分字符串

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.List;
import java.util.LinkedList;

/**
 * Demonstrate behavior of splitting strings.  The XML output created
 * here can be styled into HTML or some other useful form.
 * See poodle.xsl.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class Poodle
{
   /**
    * Generate a matrix table of how Pattern.split( ) behaves with
    * various regex patterns and limit values.
    */
   public static void main (String [] argv)
      throws Exception
   {
      String input = "poodle zoo";
      Pattern space = Pattern.compile (" ");
      Pattern d = Pattern.compile ("d");
      Pattern o = Pattern.compile ("o");
      Pattern [] patterns = { space, d, o };
      int limits [] = { 1, 2, 5, -2, 0 };

      // 如果有参数就使用提供的参数。假设参数合理。
      // 用法：输入模式[pattern ...]
      // 别忘了引用参数。
      if (argv.length != 0) {
         input = argv [0];
         patterns = collectPatterns (argv);
      }

      generateTable (input, patterns, limits);
   }

   /**
    * Output a simple XML document with the results of applying
    * the list of regex patterns to the input with each of the
    * limit values provided.  I should probably use the JAX APIs
    * to do this, but I want to keep the code simple.
    */
   private static void generateTable (String input,
      Pattern [] patterns, int [] limits)
   {
      System.out.println ("<?xml version='1.0'?>");
      System.out.println ("<table>");
      System.out.println ("\t<row>");
      System.out.println ("\t\t<head>Input: "
         + input + "</head>");
      for (int i = 0; i < patterns.length; i++) {
         Pattern pattern = patterns [i];

         System.out.println ("\t\t<head>Regex: <value>"
            + pattern.pattern(  ) + "</value></head>");
      }

      System.out.println ("\t</row>");
```

```
        for (int i = 0; i < limits.length; i++) {
            int limit = limits [i];

            System.out.println ("\t<row>");
            System.out.println ("\t\t<entry>Limit: "
                + limit + "</entry>");

            for (int j = 0; j < patterns.length; j++) {
                Pattern pattern = patterns [j];
                String [] tokens = pattern.split (input, limit);

                System.out.print ("\t\t<entry>");

                for (int k = 0; k < tokens.length; k++) {
                    System.out.print ("<value>"
                        + tokens [k] + "</value>");
                }

                System.out.println ("</entry>");
            }

            System.out.println ("\t</row>");
        }

        System.out.println ("</table>");
    }

    /**
     * If command line args were given, compile all args after the
     * first as a Pattern.  Return an array of Pattern objects.
     */
    private static Pattern [] collectPatterns (String [] argv)
    {
        List list = new LinkedList( );

        for (int i = 1; i < argv.length; i++) {
            list.add (Pattern.compile (argv [i]));
        }

        Pattern [ ] patterns = new Pattern [list.size( )];

        list.toArray (patterns);

        return (patterns);
    }
}
```

例 5-2 输出的是描述结果矩阵的 XML 文档，例 5-3 中的样式表将 XML 转换成 HTML 格式便于显示在网页浏览器上。

例 5-3. 拆分矩阵样式表

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

<!--
   XSL stylesheet to transform the simple XML output of Poodle.java
   to HTML for display in a browser.Use an XSL processor such as
   xalan with this stylesheet to convert the XML to HTML.

   @author Ron Hitchens (ron@ronsoft.com)
  -->

<xsl:output method="html"/>

<xsl:template match="/">
   <html><head><title>Poodle Zoo</title></head><body>
   <xsl:apply-templates/>
   </body></html>
</xsl:template>

<xsl:template match="table">
   <table align="center" border="1" cellpadding="5">
   <xsl:apply-templates/>
   </table>
</xsl:template>

<xsl:template match="row">
   <tr>
   <xsl:apply-templates/>
   </tr>
</xsl:template>

<xsl:template match="entry">
   <td>
   <xsl:apply-templates/>
   </td>
</xsl:template>

<xsl:template match="head">
   <th>
   <xsl:apply-templates/>
   </th>
</xsl:template>

<xsl:template match="entry/value">
   <xsl:if test="position( ) != 1">
      <xsl:text>, </xsl:text>
   </xsl:if>
   <xsl:call-template name="simplequote"/>
</xsl:template>

<xsl:template name="simplequote" match="value">
   <code>
   <xsl:text>"</xsl:text>
   <xsl:apply-templates/>
   <xsl:text>"</xsl:text>
   </code>
</xsl:template>
</xsl:stylesheet>
```

5.2.3 Matcher类　The Matcher Classic

Matcher 类为匹配字符序列的正则表达式模式提供了丰富的 API。Matcher 实例常常通过对 Pattern 对象调用 *matcher( )*方法来创建的，它常常采用由该 Pattern 封装的正则表达式：

```
package java.util.regex;

public final class Matcher
{
	public Pattern pattern( )

	public Matcher reset( )
	public Matcher reset (CharSequence input)

	public boolean matches( )
	public boolean lookingAt( )
	public boolean find( )
	public boolean find (int start)

	public int start( )
	public int start (int group)
	public int end( )
	public int end (int group)

	public int groupCount( )
	public String group( )
	public String group (int group)

	public String replaceFirst (String replacement)
	public String replaceAll (String replacement)
	public StringBuffer appendTail (StringBuffer sb)
	public Matcher appendReplacement (StringBuffer sb,
		String replacement)
}
```

Matcher 类的实例是监控状态型对象，它们封装了与特定输入字符序列匹配的具体正则表达式。Matcher 对象并不是线程安全的，因为它们在方法调用之间有保有内状态（hold internal state）。

一个 Matcher 实例来自一个 Pattern 实例，Matcher 对象的 *pattern( )*返回的是向后引用（back reference），指向创建了 Matcher 的 Pattern 对象。

Matcher 对象可以重复使用，但是因其监控状态属性，为了开始新匹配操作它们必须处于已知状态。这可通过调用 *reset( )*方法来实现，该方法在与匹配程序有关的 *CharSequence* 之前为模式匹配备好了对象。无参数的 *reset( )*将使用上次为 Matcher 设置的 *CharSequence*。如果你希望对新的字符序列进行匹配，那么你可以将一个新的 *CharSequence* 传递给 *reset( )*，随后匹配将针对目标进行。例如，随着你读取各行的文件，你可以把它传递给 *reset( )*。参见例 5-4。

例 5-4. 简单的文件 grep　Simple file grep

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

/**
 * Simple implementation of the ubiquitous grep command.
 * First argument is the regular expression to search for (remember to
 * quote and/or escape as appropriate).  All following arguments are
 * filenames to read and search for the regular expression.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class SimpleGrep
{
   public static void main (String [] argv)
      throws Exception
   {
      if (argv.length < 2) {
         System.out.println ("Usage: regex file [ ... ]");
         return;
      }

      Pattern pattern = Pattern.compile (argv [0]);
      Matcher matcher = pattern.matcher ("");

      for (int i = 1; i < argv.length; i++) {
         String file = argv [i];
         BufferedReader br = null;
         String line;

         try {
            br = new BufferedReader (new FileReader (file));
         } catch (IOException e) {
            System.err.println ("Cannot read '" + file
               + "': " + e.getMessage(  ));
            continue;
         }

         while ((line = br.readLine(  )) != null) {
            matcher.reset (line);

            if (matcher.find(  )) {
               System.out.println (file + ": " + line);
            }
         }

         br.close(  );
      }
   }
}
```

例 5-5 演示的是 *reset( )*较复杂的应用，它允许 Matcher 作用于若干不同的字符序列。

例 5-5. 提取匹配的表达式 Extracting matched expressions

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
 * Validates email addresses.
 *
 * Regular expression found in the Regular Expression Library
 * at regxlib.com.  Quoting from the site,
 * "Email validator that adheres directly to the specification
 * for email address naming.  It allows for everything from
 * ipaddress and country-code domains, to very rare characters
 * in the username."
 *
 * @author Michael Daudel (mgd@ronsoft.com) (original)
 * @author Ron Hitchens (ron@ronsoft.com) (hacked)
 */
public class EmailAddressFinder
{
    public static void main (String[] argv)
    {
        if (argv.length < 1) {
            System.out.println ("usage: emailaddress ...");
        }

        // 编译电子邮件地址检测器的模式
        Pattern pattern = Pattern.compile (
            "([a-zA-Z0-9_\\-\\.]+)@((\\[[0-9]{1,3}\\.[0-9]"
            + "{1,3}\\.[0-9]{1,3}\\.)|(([a-zA-Z0-9\\-]+\\.)+))"
            + "([a-zA-Z]{2,4}|[0-9]{1,3})(\\]?)",
            Pattern.MULTILINE);

        // 为模式制作一个Matcher对象
        Matcher matcher = pattern.matcher ("");

        // 循环遍历参数并在每个寻找地址
        for (int i = 0; i < argv.length; i++) {
            boolean matched = false;

            System.out.println ("");
            System.out.println ("Looking at " + argv [i] + " ...");

            // 将匹配器复位，查看当前实参字符串
            matcher.reset (argv [i]);

            // 遇到匹配时循环
            while (matcher.find(  ))
                // 找到一个匹配
                System.out.println ("\t" + matcher.group(  ));

                matched = true;
            }
            if ( ! matched) {
                System.out.println ("\tNo email addresses found");
            }
        }
    }
}
```

以下 EmailAddressFinder 运行一些标准地址上的输出：

```
Looking at Ron Hitchens ,ron@ronsoft.com., fred@bedrock.com,
barney@rubble.org, Wilma
<wflintstone@rockvegas.com> ...
ron@ronsoft.com
fred@bedrock.com
barney@rubble.org
wflintstone@rockvegas.com
```

下一组方法返回的是正则表达式如何适用于目标字符串的布尔标志。首先是 *matches( )*，如果整个（entire）字符序列匹配正则表达式的模式，则它返回 true。反之如果模式匹配的只是子序列，方法将返回 false。在文件中，这种方法用于选取恰好满足一定模式的行是非常有用的。这种行为（behavior）与作用于 Pattern 类的公用程序 *matches( )*相同。

*lookingAt( )*方法与 *matches( )*相似，但是它不要求整个序列的模式匹配。如果正则表达式模式匹配字符序列的 beginning（开头），则 *lookingAt( )*返回 true。*lookingAt( )*方法往往从序列的头部开始扫描。该方法的名字暗示了匹配程序正在“查看”目标是否以模式开头。如果返回为 true，那么可以调用 *start( )*、*end( )*和 *group( )*方法匹配的子序列的范围（随后将给出更多关于这些程序的内容）。

*find( )*方法运行的是与 *lookingAt( )*相同类型的匹配操作，但是它会记住前一个匹配的位置并在之后重新开始扫描。从而允许了相继调用 *find( )*对输入进行逐句比对，寻找嵌入的匹配。复位后第一次调用该方法，则扫描将从输入序列的首个字符开始。在随后调用中，它将从前一个匹配的子序列后面的第一个字符重新开始扫描。如各个调用来说，如果找到了模式将返回 true；反之将返回 false。通常你会使用 *find( )*循环访问一些文本来查找其中所有匹配的模式。

带位置参数的 *find( )*会在给定的索引位置进行隐式复位并从该位置开始扫描。然后如果需要可以调用无参数的 *find( )*扫描输入序列剩余的部分。

一旦检查到匹配，你可以通过调用 *start( )*和 *end( )*确定匹配位于字符序列的什么位置。*Start( )*方法返回的是匹配序列首个字符的索引；*end( )*方法返回的值等于匹配序列最末字符的索引加一。这些返回值与 *CharSequence.subsequence( )*的返回值一致，可直接用于提取匹配的子序列。

```
CharSequence subseq;
if (matcher.find( )) {
       subseq = input.subSequence (matcher.start(), matcher.end( ));
}
```

一些正则表达式可以匹配空字符串，这种情况下 *start( )*和 *end( )*将返回相同的值。只有当匹配之前已经过 *matches( )*、*lookingAt( )*或检测 *find( )*的检测，*start( )*和 *end( )*返回的值才有意义。如果没有检测到匹配或最后的匹配尝试返回的是 false，那么调用 *start( )*或 *end( )*将导致

java.lang.IllegalStateException（Java 语言非法状态异常）。

为了了解带有 group 参数的 *start( )*和 *end( )*，我们首先需要知道表达式捕获组（expression capture group）。（见图 5-2）

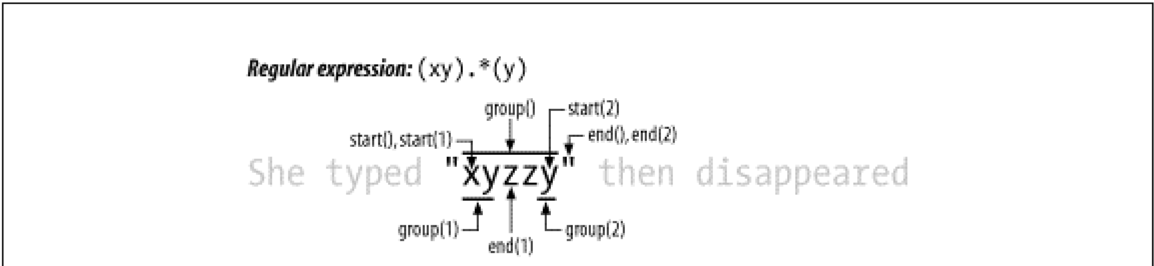


图 5-2. start( )、end( )和 group( )的值

正则表达式可能包含称为捕获组（capture group）的子表达式，它们被小括号括了起来。在正则表达式的求值期间将保存匹配这些捕获组表达式的输入子序列。一旦完全匹配操作完成，这些保存的代码片断可通过确定相应的组号从 Matcher 对象上重新获取。

捕获组可以嵌套使用，数量可以通过从左到右计算左括弧（开括号）得到。无论整个表达式是否有子组，它的捕获组总能记为组零（group zero）。例如，正则表达式 A((B)(C(D)))可能有的捕获组编号如表 5-3 所示。

5-3. A((B)(C(D)))的正则表达式捕获组	
组号	表达式组
0	A((B)(C(D)))
1	((B)(C(D)))
2	(B)
3	(C(D))
4	(D)

这种分组句法存在异常事件。以(?开头的组是个纯的（pure）或说是无法捕获的组。它的值无法保存且它对无法计算捕获组编号。（句法细节参见表 5-7。）

让我们看看方法作用于捕获组的更多细节：

```
package java.util.regex;

public final class Matcher
{
        // 这是API列表的一部分

        public int start( )
        public int start (int group)
        public int end( )
        public int end (int group)
```

```
        public int groupCount( )
        public String group( )
        public String group (int group)
}
```

捕获组在正则表达式模式中的编号由 *groupCount( )*方法返回。该值来自原始的 Pattern 对象，是不可变的。组号必须为正且小于 *groupCount( )*返回的值。传递超出范围的组号将导致 *java.lang.IndexOutOfBoundsException*（java 语言索引出界异常）。

可以将捕获组号传递给 *start( )*和 *end( )*来确定子序列是否匹配已知的捕获组子表达式。有可能出现这样一种情况，即整个表达式成功匹配但是有一个或多个的捕获组无法匹配。如果请求的捕获组当前没有设置则 *start( )*和 *end( )*方法的返回值将为-1。

（正如之前看到的）你可以利用 *start( )*和 *end( )*返回的值从输入的 *CharSequence* 中提取出匹配的子序列，但是 *group( )*方法为此提供了更简单的方式。调用带数字参数的 *group( )*将返回一个字段，该字段是匹配特殊捕获组的子序列。如果你调用的 *group( )*不含参数，则返回将是与整个正则表达式（组零）匹配的子序列。代码如下：

```
String match0 =
     input.subSequence (matcher.start(), matcher.end()).toString( );
String match2 = input.subSequence (matcher.start (2),
     matcher.end (2)).toString( );
```

上述代码与下列代码等效：

```
String match0 = matcher.group( );
String match2 = matcher.group(2);
```

最后让我们看看 Matcher 对象解决修改字符序列的方法。正则表达式最常见的应用之一是查找并替换（search-and-replace）。这种应用使用 *replaceFirst( )*和 *replaceAll( )*可以轻轻松松就搞定。它们的行为方式是相同的，区别在于 *replaceFirst( )*在找到第一个匹配后就会停止，而 *replaceAll( )*将循环执行直到替换完所有的匹配。二者都带有 String 参数，String 参数是用于替换输入字符序列中匹配模式的替换值（replacement value）。

```
package java.util.regex;

public final class Matcher
{
        // 这是API部分列表

        public String replaceFirst (String replacement)
        public String replaceAll (String replacement)
}
```

上文提过，捕获组在正则表达式内可以向后引用（back-reference）。它们也可以被你提供组 *replaceFirst( )*或 *replaceAll( )*的替换字符串引用。捕获组号通过添加美元符号$可嵌入替换字符串中。当替换字符串被替换成结果字符串时，每次出现的$g 将被 *group( )*返回的值代替。如

果你想在替换字符串使用字面量（literal）美元符号，那么你必须在它前面加个反斜杠符号（\$）。如果想要传递反斜杠符号，你必须多加一个反斜杠（\\）。如果你想在捕获组引用后面跟上字面量的数值型数字，那么你可以用反斜杠将它们与组号分开，像这样：123$2\456。表 5-4 对此给出了一些例子。示例代码见例 5-6。

表 5-4. 匹配模式的替换

正则模式	输入	替换	replaceFirst( )	replaceAll( )
a*b	aabfooaabfooabfoob	-	-fooaabfooabfoob	-foo-foo-foo-
\p{Blank}	fee fie foe fum		fee_fie foe fum	fee fie foe fum
([bB])yte	Byte for byte	$1ite	Bite for byte	Bite for bite
\d\d\d\d([-])	card #1234-5678-1234	xxxx$1	card #xxxx-5678-1234	card #xxxx-xxxx-1234
(up|left)(*)(right|down)	left right, up down	$3$2$1	right left, up down	right left, down up
([CcPp][hl]e[ea]se)	I want cheese. Please.	 $1 	I want cheese . Please.	I want cheese . Please .

例 5-6. 正则表达式替换 Regular expression replacement

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
* Exercise the replacement capabilities of the java.util.regex.Matcher
 * class. Run this code from the command line with three or more arguments.
 * 1) First argument is a regular expression
 * 2) Second argument is a replacement string, optionally with capture
 *    group references ($1, $2, etc)
 * 3) Any remaining arguments are treated as input strings to which the
 *    regular expression and replacement strings will be applied.
 * The effect of calling replaceFirst( ) and replaceAll( ) for each input
 * string will be listed.
 *
 * Be careful to quote the commandline arguments if they contain spaces
 * or special characters.
*
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class RegexReplace
{
    public static void main (String [] argv)
    {
        // 完整性检查，至少需要三个参数
        if (argv.length < 3) {
            System.out.println ("usage: regex replacement input ...");
            return;
        }

        // 用助词符号名保存正则及替换字符串
```

```
        String regex = argv [0];
        String replace = argv [1];

        // 编译表达式；一次只能编译一个
        Pattern pattern = Pattern.compile (regex);
        // 得到Matcher实例，暂时先使用虚设的输入字符串
        Matcher matcher = pattern.matcher ("");

        // 打印输出用于参考
        System.out.println (" regex: '" + regex + "'");
        System.out.println ("   replacement: '" + replace + "'");

        // 对各个剩余的参数字符串应用正则/替换
        for (int i = 2; i < argv.length; i++) {
            System.out.println ("------------------------");

            matcher.reset (argv [i]);

            System.out.println ("        input: '"
                + argv [i] + "'");
            System.out.println ("replaceFirst(  ): '"
                + matcher.replaceFirst (replace) + "'");
            System.out.println ("  replaceAll(  ): '"
                + matcher.replaceAll (replace) + "'");
        }
    }
}
```

下列是运行 RegexReplace 后的输出结果：

```
        regex: '([bB])yte'
   replacement: '$1ite'
------------------------
        input: 'Bytes is bytes'
replaceFirst(  ): 'Bites is bytes'
  replaceAll(  ): 'Bites is bites'
```

记住:正则表达式会在你提供的字符串中翻译反斜杠。另外在字面量的 String 中，Java 编译器要求各个反斜杠需要有两个反斜杠，即如果你想在正则中转义（escape）一个反斜杠），那么你在编译过的 String 需要中需要两反斜杠。如果在编译的正则字符串中需要两个连续的反斜杠，那么在 Java 源代码中需要四个连接的反斜杠。

为了生成 a\b 的替换序列，*replaceAll( )*的 String 字面量参数必须是 a\\\\b（见例 5-7）。统计这些反斜杠时千万要小心啊！

例 5-7. 正则表达式中的反斜杠 Backslashes in regular expressions

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
 * Demonstrate behavior of backslashes in regex patterns.
 *
```

```
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class BackSlashes
{
    public static void main (String [] argv)
    {
        // 在输入中把“a\b”替换成XYZ或ABC
        String rep = "a\\\\b";
        String input = "> XYZ <=> ABC <";
        Pattern pattern = Pattern.compile ("ABC|XYZ");
        Matcher matcher = pattern.matcher (input);

        System.out.println (matcher.replaceFirst (rep));
        System.out.println (matcher.replaceAll (rep));

        // 在输入中更改所有的新行来转义， DOS-like CR/LF
        rep = "\\\\r\\\\n";
        input = "line 1\nline 2\nline 3\n";
        pattern = Pattern.compile ("\\n");
        matcher = pattern.matcher (input);

        System.out.println ("");
        System.out.println ("Before:");
        System.out.println (input);

        System.out.println ("After (dos-ified, escaped):");
        System.out.println (matcher.replaceAll (rep));
    }
}
```

下列是运行*BackSlashes*的输出结果：

```
> a\b <=> ABC <
> a\b <=> a\b <

Before:
line 1
line 2
line 3

After (dos-ified, escaped):
line 1\r\nline 2\r\nline 3\r\n
```

Matcher API 列出了两个追加方法，它们在循环访问输入字符序列时很有用，它们重复调用 *find( )*。

```
package java.util.regex;

public final class Matcher
{
        // 这是部分的API列表

        public StringBuffer appendTail (StringBuffer sb)
        public Matcher appendReplacement (StringBuffer sb,
String replacement)
```

追加方法不是返回已经运行过替换的新 String 而是添加到你提供的 StringBuffer 对象上。这就允许你决定在找到匹配的各点是否进行替换或计算与多个输入字符串的匹配结果。通过 *appendReplacement( )*和 *appendTail( )*，你可以全面控制查找和替换过程。

Matcher 对象记住的状态信息位（the bits of state information）之一是追加位置（append position）。追加位置是用于记住输入字符序列的量，这些字符序列已经通过之前调用 *appendReplacement( )*复制了出来。当调用 *appendReplacement( )*时，将发生如下过程：

1. 从输入中读取字符是从当前追加位置开始，读取的字符将被添加到已知的 StringBuffer 中。最后复制的字符就在匹配模式的首个字符之前。这个字符位于 *start( )*返回的索引减一的位置。

2. 如先前描述的，替换字符串被添加给 StringBuffer 并替换任何嵌入的捕获组引用。

3. 追加位置更新成跟在匹配模式后面的字符的索引，这个索引是 *end( )*返回的值。

仅当前一个匹配操作成功（通常调用 *find( )*）*appendReplacement( )*方法才能正常工作。如果前一个匹配返回的是 false 或在复位后立即调用该方法，你将得到一个“令人愉快的奖励”：*java.lang.IllegalStateException*（java 语言非法状态异常）。

但是别忘了：在输入中除了最后的模式匹配外还有剩余的字符。你很可能不想失去它们，但是 *appendReplacement( )*不会复制它们，并且在 *find( )*无法找到更多的匹配后 *end( )*将不会返回有用的值。这种情况下 *appendTail( )*方法正好可以复制输入中余下的部分。它仅是复制了从当前追加位置到输入结果的所有字符，并把它们追加到给定的 StringBuffer 中。下列代码是 *appendReplacement( )*和 *appendTail( )*典型的使用情况。

```
Pattern pattern = Pattern.compile ("([Tt])hanks");
Matcher matcher = pattern.matcher ("Thanks, thanks very much");
StringBuffer sb = new StringBuffer(  );
while (matcher.find(  )) {
      if (matcher.group(1).equals ("T")) {
            matcher.appendReplacement (sb, "Thank you");
      } else {
            matcher.appendReplacement (sb, "thank you");
      }
}
matcher.appendTail (sb);
```

表 5-5 显示的是上述代码应用到 StringBuffer 上后变化的序列。

表 5-5. 使用 appendReplacement( )和 appendTail( )		
追加位置	执行	生成的 StringBuffer
0	appendReplacement (sb, "Thank you")	Thank you
6	appendReplacement (sb, "thank you")	Thank you, thank you
14	appendTail (sb)	Thank you, thank you very much

经过追加操作的该序列生成的 StringBuffer 对象 sb 包含字符串“Thank you, thank you very much”。例 5-8 是个完整的代码示例，它显示了这种替换类型及运行相同替换的代替方式。在

这个简单例子里，可以使用捕获组的值，因为匹配模式的首个字母与替换的首个字母相同。在更复杂的例子中，输入与替换值间可能不存在重叠部分（overlap）。利用 *Matcher.find( )*和*Matcher.appendReplacement( )*，你可以通过编程方式调解每个替换，从而可能随时在各个点引入不同的替换值。

例 5-8. 正则表达式追加/替换 Regular expression append/replace

```
package com.ronsoft.books.nio.regex;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
 * Test the appendReplacement() and appendTail(  ) methods of the
 * java.util.regex.Matcher class.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class RegexAppend
{
    public static void main (String [] argv)
    {
        String input = "Thanks, thanks very much";
        String regex = "([Tt])hanks";
        Pattern pattern = Pattern.compile (regex);
        Matcher matcher = pattern.matcher (input);
        StringBuffer sb = new StringBuffer( );
        //循环直到遇到匹配
        while (matcher.find( )) {
            if (matcher.group(1).equals ("T")) {
                matcher.appendReplacement (sb, "Thank you");
            } else {
                matcher.appendReplacement (sb, "thank you");
            }
        }

        // 完成到StringBuffer的传送
        matcher.appendTail (sb);

        // 打印结果
        System.out.println (sb.toString( ));
        // 让我们再试试在替换中使用$n转义
        sb.setLength (0);
        matcher.reset(  );

        String replacement = "$1hank you";
        //循环直到遇到匹配
        while (matcher.find( )) {
            matcher.appendReplacement (sb, replacement);
        }

        // 完成到StringBuffer传送
        matcher.appendTail (sb);

        // 打印结果
        System.out.println (sb.toString( ));

        // 再来一次，简单的方法（因为这个父子很简单）
```

```
        System.out.println (matcher.replaceAll (replacement));

        // 最后一次，只使用字符串
        System.out.println (input.replaceAll (regex, replacement));

    }
}
```

5.3 String类的正则表达式方法 Regular Expression Methods of the String Class

从前几个章节来看，字符串和正则表达式关联密切是显而易见的。这是很自然，我们的“老朋友”Sting 类已经添加了一些公用程序来执行常见的正则表达式操作：

```
package java.lang;

public final class String
        implements java.io.Serializable, Comparable, CharSequence
{
        // 这是部分的API列表

        public boolean matches (String regex)
        public String [] split (String regex)
        public String [] split (String regex, int limit)
        public String replaceFirst (String regex, String replacement)
        public String replaceAll (String regex, String replacement)
}
```

所有新的 String 方法是 Pattern 或 Matcher 类的传递调用（pass-through call）。现在你知道了 Pattern 和 Matcher 是如何使用的，并且利用这些 String 公用程序的交互操作（interoperate）应当是傻瓜式的。表 5-6 对这些方法作了总结，不再对方法一一赘述。

表 5-6. String 类的正则表达式方法

字符串方法签名	等效的 java.util.regex
input.matches (String regex)	Pattern.matches (String regex, CharSequence input)
input.split (String regex)	pat.split (CharSequence input)
input.split (String regex, int limit)	pat.split (CharSequence input, int limit)
input.replaceFirst (String regex, String replacement)	match.replaceFirst (String replacement)
input.replaceAll (String regex, String replacement)	match.replaceAll (String replacement)

表 5-6 中，假定 String 对象名为 input，Pattern 对象名为 pat，而 Matcher 对象名为 match:

```
String input = "Mary had a little lamb";
String [] tokens = input.split ("\\s+");   // 在空白符上拆分
```

在 JDK 1.4 之前，没有一个正则表达式公用程序缓存任何表达式或进行其它优化。一些 JVM 实现可能选择缓存或重用模式对象，但是它们却不足以采信。如果你希望重复应用相同的模式匹配操作，那么使用 java.util.regex 中的类更为高效。

5.4 Java正则表达式句法　Java Regular Expression Syntax

下面是 java.util.regex 包支持的正则表达式句法的总结。Java 世界“瞬息万变”，因此你需要时常检查一下你正在使用的 Java 实现提供的当前文档。这里提供的信息为你提供了迈出第一步的快速参考。

java.util.regex 类可全面感知 Unicode（fully Unicode-aware），它们完全遵循《Unicode 技术报告#18：Unicode 正则表达式指南》中的准则。文章参见：

http://www.unicode.org/unicode/reports/tr18。

之前提过，java.util.regex 的句法与 Perl 相似，却又不尽相同。java.util.regex 缺少的是能够在表达式中嵌入 Perl 代码这一主要功能（这要求插入完整的 Perl 解译器）。Java 句法中添加的重要功能是占有型量词（possessive quantifier），它们比常规的贪婪量词（greedy quantifier）还要“贪婪”。占有型量词会尽可能地多匹配目标，即便这意味着表达式剩余的部分无法匹配。Java 正则表达式也支持一些 Perl 不支持的 Unicode 转义序列。java.util.regex.Pattern 完整详情请参阅 Javadoc 页面。

表 5-7 是正则表达式一览表。它复制的是《Java In A Nutshell》第四版（O’Reilly）。

表 5-7. Java 正则表达式句法一览表

句法	匹配
单字符	
X	字符x，由于x并不是标点符号字符，它在正则表达式句法中没有特殊的意义
\p	标点符号（punctuation）字符p。
\\	反斜杠字符。
\n	换行字符，等效于\u000A。
\t	制表符，等效于\u0009。
\r	回车字符，等效于\u000D。
\f	换页符，等效于\u000C。
\e	转义符/换码符，等效于\u001B。
\a	报警字符，等效于\u0007。
\uhhhh	十六进制码为hhhh的Unicode字符。
\xhh	十六进制码为hh的字符。
\0n	八进制码为n的字符。
\0nn	八进制码为nn的字符。
\0nnn	八进制码为nnn的字符，其中nnn<=377（译注[15]）
\cx	由x指明的控制符^x。
字符组（Character classes）	
[...]	中括号内的字符之一。字符可能指定为字面量，句法也允许采用交集、并集和差集运算符指定字符范围。见下列的具体例子。
[^...]	除中括号内字符外的任意字符
[a-z0-9]	字符范围：在 a 和 z 之间或 0 和 9 之间的字符。包括两头的字符 a、z、0、9。
[0-9[a-fA-F]]	并集：与[0-9a-fA-F]等价。
[a-z&&[aeiou]]	交集：等价于[aeiou]。
[a-z&&[^aeiou]]	差集：a 到 z 中除元音之外的字符. 译注：26 个英文字母中只有 5 个元音 aeiou，故有上面一说。
.	除行终止符外的任意字符。如果设置了 DOTALL 标志，它将匹配包括行终止符在内的任意字符。
\d	匹配 ASCII 数字：[0-9]。
\D	匹配非 ASCII 数字的字符：[^\d]。
\s	匹配 ASCII 空白字符：[\t\n\f\r\x0B]。

15： Java 中有 3 种方法表示整数。十进制（0-9），int i = 99；八进制（0-7），在数的前面放置一个零，int five = 05，int nine = 011；十六进制（0~9,a~f），在数前放置 0x 或者 0X，int x = 0x0001，int y = 0xb，int z = 0x7FFff

\S	匹配任何非 ASCII 空白字符的字符：[^\s]。
\w	匹配包括下划线在内的 ASCII 单词字符：[a-zA-Z0-9_]
\W	匹配除 ASCII 单词字符外的任意字符：[^\w]。
\p{group}	在指定组中的任意字符。组名见下面内容。许多组名来自 POSIX，这就是 p 用作该字符类标识的原因。
\P{group}	不在指定组内的任意字符。译注：原文有误。
\p{Lower}	ASCII 小写字母：[a-z]。
\p{Upper}	ASCII 大写字母：[A-Z]。
\p{ASCII}	任意的 ASCII 字符：[\x00-\x7f]
\p{Alpha}	ASCII 字母：[a-zA-Z]
\p{Digit}	ASCII 数字：[0-9]
\p{XDigit}	十六进制数字：[0-9a-fA-F]。
\p{Alnum}	ASCII 字母或数字：[\p{Alpha}\p{Digit}]
\p{Punct}	ASCII 标点符号：!"#$%&'( )*+,-./:;<=>?@[\]^_`{|}~]其中之一。
\p{Graph}	可见的 ASCII 字符：[\p{Alnum}\p{Punct}]
\p{Print}	可见的 ASCII 字符：等效于\{Graph}。译注：区别于空白符
\p{Blank}	ASCII 空格符或制表符：[\t]。
\p{Space}	ASCII 空白符：[\t\n\f\r\x0b]。
\p{Cntrl}	ASCII 控制符：[\x00-\x1f\x7f]。
\p{category}	指定 Unicode 类别内的任意字符。类别名是 Unicode 标准定义的单字母或双字母代码。单字母代码包括：L 表示字母 letter，N 表示数值 number，S 表示标志 symbol，S 表示分隔符 separator，P 表示标点符号 punctuation。双字母代码代表的子类别如：Lu 代表大写字母 uppercase letter，Nd 代表十进制数字 decimal digit，Sc 代表货币符号 currency symbol，Sm 代表数学符号 math symbol 及 Zs 代表空格 space separator。与这些子类别对应的常量集请参见 java.lang.Character，需要提醒的是本书中单字母及双字母代码并不完整。
\p{block}	指定 Unicode 块中的任意字符。在 Java 正则表达式中，块名以“In”开始，随后是 Unicode 块名，块名是大小写混合并以大写开头的，不含空格或下划线。例如：\p{InOgham}或\p{InMathematicalOperators}。Unicode 块名的列表见 java.lang.Character.UnicodeBlock。
序列、等效表示、组及引用	
xy	先匹配 x 接着匹配 y。
x|y	匹配 x 或 y。

(...)	分组。圆括号内组的子表达式可用*、+、?、|等分成各个单元。此外还可以“捕捉”匹配该组的字符供以后使用。
(?:...)	只是分组。组的子表达式用( )括起来，但是不捕捉匹配的文本。
\n	当捕获的组号 n 首次匹配时，对相同的字符进行匹配。注意当 n 后面跟另一个数字时，须使用最大的值，因为那才是有效组数。
重复 （注 3）	
x?	允许 x 出现零次或一次，例如 x 是可选的。
x*	允许 x 出现零次或多次。
x+	允许 x 出现一次或多次。
x{n}	x 恰好出现 n 次。
x{n,}	x 出现 n 次或更多次。
x{n,m}	x 至少出现 n 次，最多出现 m 次。
锚定符 （注 4）	
^	匹配输入字符串的起始，或者如果指定 MULTILINE 标志，则可以匹配字符串或任意新行的起始。
$	匹配输入字符串的结尾，或者如果指定 MULTILINE 标志，则可以匹配字符串的结尾或字符串内行的结尾。
\b	单词边界：字符串中单词与非单词字符之间的位置。
\B	字符串中不是单词边界的位置。
\A	输入字符串的起始。与^类似，但是无论设置什么标志，始终不匹配新行的起始。
\Z	输入字符串的结束，忽略后面的行终止符。
\z	输入字符串的结束，包括行终止符。
\G	前一个匹配的结束。
(?=x)	向前的正断定（positive look-ahead assertion）。要求后面字符匹配 x，但是不包括匹配中的字符。
(?!x)	向前的负断定（negative look-ahead assertion）。要求后面的字符不匹配模式 x。
(?<=x)	向后的正断定（positive look-behind assertion）。要求紧挨在位置

注 3： 这些重复的字符被认为是贪婪量词，因为在尽可能多地匹配 x 的同时，它们仍旧允许剩下的正则表达式进行匹配。如果你想要个“勉强量词 reluctant quantifier”，即尽可能少地匹配 x 的同时仍旧允许剩下的正则表达式进行匹配，在之前的量词后面加个问号即可。例如，用*?代替*，用{2,}?代替{2,}。或者，如果你在量词后面跟一个加号而不是问号，那么你指定了一个“占有量词”，它尽可能多地进行匹配，但是剩余的正则表达式将无法匹配。当你确定占有量词无损剩余的匹配时，它们是很有用的，因为它们比常规的贪婪量词更高效。（关于三者量词的区别个人喜欢博文 http://qq8903239.blog.163.com/blog/static/35074384201081345728809/中的解释）

注 4：锚定符不是匹配字符而是匹配字符间的零宽位置，将匹配“锚定”在特定的位置。

	前面的字符匹配 x，但不包括匹配中的字符。x 必须是个字符数一定的模式。
(?<!x)	向后的负断定（negative look-behind assertion）。要求紧挨在位置前面的字符不匹配 x。x 必须是字符数一定的模式。
其它	
(?>x)	x 与表达式的剩余部分无关，不考虑匹配是否会导致表达式剩余部分无法匹配。这在优化某些复杂的正则表达式时十分管用。这种形式的组不捕捉匹配的文本。
(?onflags-offflags)	不匹配任何东西，但是由 onflags 来打开标志，并且由 offflags 来禁用标志。这两个字符串以下面的字母任何组合，与下列的 Pattern 常量对应：i (CASE_INSENSITIVE)、d (UNIX_LINES)、m (MULTILINE)、s (DOTALL)、u (UNICODE_CASE)，和 x (COMMENTS)。以这种方式指定的标志设置影响了它们在表达式中的出现和，影响持续到表达式结束，或括入括号的组（它们是一个组成部分）的结束，或直到它们被另一个标志设置表达式覆盖。
(?onflags-offflags:x)	匹配 x，只将特定标志应用在该子表达式上。这是无法捕捉的组，例如(?:...)，其中有附加标志。
\Q	不匹配任何字符，但是引用随后的模式文本直到\E。在引用部分的所有字符都按字面量翻译进行匹配，（除\E 外）不具任何特殊含义。
\E	不匹配任何字符；结束以\Q 开始的引用。
#comment	如果设置了 COMMENT 标志，在 a、#和行结尾之间的模式文本被认为是注释并忽略。

5.5 面向对象的文件Grep　An Object-Oriented File Grep

例 5-9 实现了一个面向熟悉的 *grep* 命令的对象。*Grep* 类的实例包含正则表达式，可用于浏览相同模式的不同文件。*Grep.grep( )*程序的结果是类型安全的 *Grep.MatchedLine* 对象阵列。*MatchedLine* 类包含在 *Grep* 中的类。你必须将它作为 *Grep.MatchedLine* 引用或将它单独导入。

例 5-9. 面向对象的 grep　Object-oriented grep

```
package com.ronsoft.books.nio.regex;

import java.io.File;
import java.io.FileReader;
import java.io.LineNumberReader;
import java.io.IOException;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * A file searching class, similar to grep, which returns information
 * about lines matched in the specified files.  Instances of this class
 * are tied to a specific regular expression pattern and may be applied
 * repeatedly to multiple files.  Instances of Grep are thread safe, they
 * may be shared.
 *
 * @author Michael Daudel (mgd@ronsoft.com) (original)
 * @author Ron Hitchens (ron@ronsoft.com) (hacked)
 */
public class Grep
{
    // 用于该实例的模式
    private Pattern pattern;

    /**
     * Instantiate a Grep object for the given pre-compiled Pattern object.
     * @param pattern A java.util.regex.Pattern object specifying the
     *  pattern to search for.
     */
    public Grep (Pattern pattern)
    {
        this.pattern = pattern;
    }

    /**
     * Instantiate a Grep object and compile the given regular expression
     * string.
     * @param regex The regular expression string to compile into a
     *  Pattern for internal use.
     * @param ignoreCase If true, pass Pattern.CASE_INSENSITIVE to the
     *  Pattern constuctor so that seaches will be done without regard
     *  to alphabetic case.  Note, this only applies to the ASCII
     *  character set.  Use embedded expressions to set other options.
     */
    public Grep (String regex, boolean ignoreCase)
    {
```

```
      this.pattern = Pattern.compile (regex,
         (ignoreCase) ? Pattern.CASE_INSENSITIVE : 0);

   }
   /**
    * Instantiate a Grep object with the given regular expression string,
    * with default options.
    */
   public Grep (String regex)
   {
      this (regex, false);
   }

   // ---------------------------------------------------------------

   /**
    * Perform a grep on the given file.
    * @param file A File object denoting the file to scan for the
    *  regex given when this Grep instance was constructed.
    * @return A type-safe array of Grep.MatchedLine objects describing
    *  the lines of the file matched by the pattern.
    * @exception IOException If there is a problem reading the file.
    */
   public MatchedLine [] grep (File file)
      throws IOException
   {
      List list = grepList (file);
      MatchedLine matches [] = new MatchedLine [list.size(  )];

      list.toArray (matches);

      return (matches);
   }

   /**
    * Perform a grep on the given file.
    * @param file A String filename denoting the file to scan for the
    *  regex given when this Grep instance was constructed.
    * @return A type-safe array of Grep.MatchedLine objects describing
    *  the lines of the file matched by the pattern.
    * @exception IOException If there is a problem reading the file.
    */
   public MatchedLine [] grep (String fileName)
      throws IOException
   {
      return (grep (new File (fileName)));
   }

   /**
    * Perform a grep on the given list of files.  If a given file cannot
    * be read, it will be ignored as if empty.
    * @param files An array of File objects to scan.
    * @return A type-safe array of Grep.MatchedLine objects describing
    *  the lines of the file matched by the pattern.
    */
   public MatchedLine [] grep (File [] files)
   {
      List aggregate = new LinkedList(  );

      for (int i = 0; i < files.length; i++) {
         try {
            List temp = grepList (files [i]);

            aggregate.addAll (temp);
         } catch (IOException e) {
            // 忽略I/O异常
```

```java
        }
    }

    MatchedLine matches [] = new MatchedLine [aggregate.size(  )];

    aggregate.toArray (matches);

    return (matches);
}

// -------------------------------------------------------------

/**
 * Encapsulation of a matched line from a file.  This immutable
 * object has five read-only properties:<ul>
 * <li>getFile(  ): The File this match pertains to.</li>
 * <li>getLineNumber(  ): The line number (1-relative) within the
 *   file where the match was found.</li>
 * <li>getLineText(  ): The text of the matching line</li>
 * <li>start(  ): The index within the line where the matching
 *  pattern begins.</li>
 * <li>end(  ): The index, plus one, of the end of the matched
 *  character sequence.</li>
 * </ul>
 */
public static class MatchedLine
{
    private File file;
    private int lineNumber;
    private String lineText;
    private int start;
    private int end;

    MatchedLine (File file, int lineNumber, String lineText,
        int start, int end)
    {
        this.file = file;
        this.lineNumber = lineNumber;
        this.lineText = lineText;
        this.start = start;
        this.end = end;
    }

    public File getFile(  )
    {
        return (this.file);
    }

    public int getLineNumber(  )
    {
        return (this.lineNumber);
    }

    public String getLineText(  )
    {
        return (this.lineText);
    }
    public int start(  )
    {
        return (this.start);
    }

    public int end(  )
    {
        return (this.end);
    }
```

```
}

// ------------------------------------------------------------

/**
 * Run the grepper on the given File.
 * @return A (non-type-safe) List of MatchedLine objects.
 */
private List grepList (File file)
    throws IOException
{
    if ( ! file.exists(  )) {
        throw new IOException ("Does not exist: " + file);
    }
    if ( ! file.isFile(  )) {
        throw new IOException ("Not a regular file: " + file);
    }
    if ( ! file.canRead(  )) {
        throw new IOException ("Unreadable file: " + file);
    }

    LinkedList list = new LinkedList(  );
    FileReader fr = new FileReader (file);
    LineNumberReader lnr = new LineNumberReader (fr);
    Matcher matcher = this.pattern.matcher ("");
    String line;

    while ((line = lnr.readLine(  )) != null) {
        matcher.reset (line);

        if (matcher.find(  )) {
            list.add (new MatchedLine (file,
                lnr.getLineNumber(  ), line,
                matcher.start(), matcher.end(  )));
        }
    }

    lnr.close(  );

    return (list);
}
// ----------------------------------------------------------------

/**
 * Test code to run grep operations.  Accepts two command-line
 * options: -i or --ignore-case, compile the given pattern so
 * that case of alpha characters is ignored.  Or -1, which runs
 * the grep operation on each individual file, rather that passing
 * them all to one invocation.  This is just to test the different
 * methods.  The printed ouptut is slightly different when -1 is
 * specified.
 */
public static void main (String [] argv)
{
    // 设置默认值
    boolean ignoreCase = false;
    boolean onebyone = false;
    List argList = new LinkedList(  );    // 采集变量

    // 循环遍历变量，查找转换并保存模式及文件名
    for (int i = 0; i < argv.length; i++) {
        if (argv [i].startsWith ("-")) {
            if (argv [i].equals ("-i")
                || argv [i].equals ("--ignore-case"))
            {
                ignoreCase = true;
            }
```

```
        if (argv [i].equals ("-1")) {
            onebyone = true;
        }
        continue;
    }
    // 不是转移（switch），将其添加到列表中
    argList.add (argv [i]);
}
// 是否有足够的变量可以运行？
if (argList.size(  ) < 2) {
    System.err.println ("usage: [options] pattern filename ...");
    return;
}
// 列表中第一个变量将被作为正则模式。
// 将模式及忽略大小写标志的当前值传递给新的Grep对象。
Grep grepper = new Grep ((String) argList.remove (0),
    ignoreCase);
// 随意点，拆分成调用grep程序和打印结果两种方式
if (onebyone) {
    Iterator it = argList.iterator(  );

    // 循环遍历文件名并用grep处理它们
    while (it.hasNext(  )) {
        String fileName = (String) it.next(  );

        // 在每次grep前先打印文件名
        System.out.println (fileName + ":");

        MatchedLine [] matches = null;
        // 捕获异常
        try {
            matches = grepper.grep (fileName);
        } catch (IOException e) {
            System.err.println ("\t*** " + e);
            continue;
        }

        // 打印匹配行的资料
        for (int i = 0; i < matches.length; i++) {
            MatchedLine match = matches [i];

            System.out.println ("  "
                + match.getLineNumber(  )
                + " [" + match.start(  )
                + "-" + (match.end(  ) - 1)
                + "]: "
                + match.getLineText(  ));
        }
    }
} else {
    // 把文件名列表转换到File阵列中
    File [] files = new File [argList.size(  )];

    for (int i = 0; i < files.length; i++) {
        files [i] = new File ((String) argList.get (i));
    }
    // 运行grep程序；忽略无法读取的文件
    MatchedLine [] matches = grepper.grep (files);

    // 打印匹配行的资料
```

```
            for (int i = 0; i < matches.length; i++) {
                MatchedLine match = matches [i];

                System.out.println (match.getFile().getName(  )
                    + ", " + match.getLineNumber(  ) + ": "
                    + match.getLineText(  ));
            }
        }
    }
}
```

5.6 总结 Summary

本章中我们讨论了期待已久的在 1.4 版本中添加到 J2SE 平台上的正则表达式类：

CharSequence

在 5.2.1 节中我们了解了新的 *CharSequence* 接口，知道了它用若干个类实现了抽象地描述字符序列。

Pattern

Pattern 类把正则表达式封装在不变的对象实例中。在 5.2.2 节我们看到了 Pattern 的 API 并学会了如何通过编译表达式字符串来创建实例。我们还知道了用于进行单次匹配的一些静态实用程序。

Matcher

Matcher 类是状态机对象，它在输入字符序列上使用了 Pattern 对象来寻找输入中匹配的模式。5.2.3 节描述了 Matcher API，包括如何在 Pattern 对象上创建新的 Matcher 实例及如何运行各种类型的匹配操作。

String

1.4 版本中添加的 String 类有了一些新的正则表达式公用程序。5.3 节对此作了总结。

java.util.regex.Pattern 支持的正则表达式句法已在表 5-7 中列出。其句法与 Perl 5 很接近。

现在我们将这次游历划上圆满的句号。在下一章中，你将领略字符集这一充满“异国情调”并且有时神秘的世界。

第六章 字符集

Here, put this fish in your ear. (在这，把鱼放到您的耳朵里。)

--Ford Prefect

我们生活在一个变化莫测的世界中。甚至在这个我们称之为地球的平凡的 M 级行星上，我们也使用数百种不同的语言。在《The Hitchhikers Guide to the Galaxy》（即《银河系漫游指南》）中，Arthur Dent 把 Babelfish（宝贝鱼）放在耳朵里，从而解决了他的语言问题。之后在他偶然的银河旅行[1]中，他就可以理解所遇到的由不同字符（至少可以说）组成的语言。

在 Java 平台上，我们没有奢侈的 Babelfish 技术（至少现在没有）[2]，但我们仍必须处理多种语言以及组成这些语言的多个字符。幸运的是，Java 是第一个被广泛使用的编程语言，它使用内在的 Unicode 来表示字符。与以字节为导向的编程语言例如 C 或 C++相比，Unicode 的固有支持大大的简化了字符数据处理，但决不是自动的处理字符。您仍需要理解字符映射的工作原理以及如何处理多个字符集。

6.1 字符集基础

在讨论 `java.nio.charsets` 中新类的细节之前，让我们先来定义一些与字符集和字符代码转换相关的术语。新的字符集类表示到该领域的更标准化的方法，所以明确术语的使用是很重要的。

Character set（字符集）

字符的集合，也就是，带有特殊语义的符号。字母“A”是一个字符。“%”也是一个字符。没有内在数字价值，与 ASCII，Unicode，甚至是电脑也没有任何的直接联系。在电脑产生前的很长一段时间内，符号就已经存在了。

Coded character set（编码字符集）

一个数值赋给一个字符的集合。把代码赋值给字符，这样它们就可以用特定的字符编码集表达数字的结果。其他的编码字符集可以赋不同的数值到同一个字符上。字符集映射通常是由标准组织确定的，例如 USASCII，ISO 8859-1，Unicode (ISO 10646-1)，以及 JIS X0201。

Character-encoding scheme（字符编码方案）

编码字符集成员到八位字节（8 bit 字节）的映射。编码方案定义了如何把字符编码的序列表达为字节序列。字符编码的数值不需要与编码字节相同，也不需要是一对一或一对多个的关系。原则上，把字符集编码和解码近似视为对象的序列化和反序列化。

通常字符数据编码是用于网络传输或文件存储。编码方案不是字符集，它是映射；但

1 他不设法阻止地球毁灭，但是这与主题不相关。

2 通过 http://babelfish.altavista.com/访问。

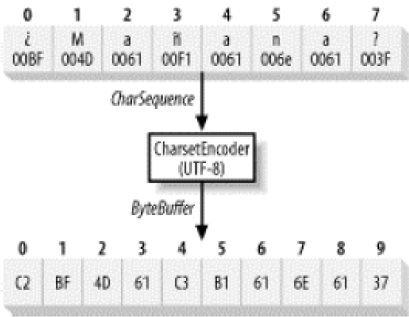
是因为它们之间的紧密联系，大部分编码都与一个独立的字符集相关联。例如，UTF-8，仅用来编码 Unicode 字符集。尽管如此，用一个编码方案处理多个字符集还是可能发生的。例如，EUC 可以对几个亚洲语言的字符进行编码。

图 6-1 是使用 UTF-8 编码方案将 Unicode 字符序列编码为字节序列的图形表达式。UTF-8 把小于 `0x80` 的字符代码值编码成一个单字节值（标准 ASCII）。所有其他的 Unicode 字符都被编码成 2 到 6 个字节的多字节序列(http://www.ietf.org/rfc/rfc2279.txt)。

Charset（字符集）

> 术语 *charset* 是在 RFC2278(http://ietf.org/rfc/rfc2278.txt)中定义的。它是编码字符集和字符编码方案的集合。`java.nio.charset` 包的锚类是 *Charset*，它封装字符集抽取。

图 6-1. 字符编码成 UTF-8.



Unicode 是 16-位字符编码。[3] 它试着把全世界所有语言的字符集统一到一个独立的、全面的映射中。它赢得了一席之地，但是目前仍有许多其他字符编码正在被广泛的使用。大部分的操作系统在 I/O 与文件存储方面仍是以字节为导向的，所以无论使用何种编码，Unicode 或其他编码，在字节序列和字符集编码之间仍需要进行转化。

由 `java.nio.charset` 包组成的类满足了这个需求。这不是 Java 平台第一次处理字符集编码，但是它是最系统、最全面、以及最灵活的解决方式。

`java.nio.charset.spi` 包提供服务器供给接口（SPI），使编码器和解码器可以根据需要选择插入。

6.2 字符集

到 JDK1.4 为止，每个 JVM 实现都需要支持标准字符集，表 6-1 列出了具体内容。JVM 实现可以随意的支持额外的字符集，但是必须提供这个最大集。（以您的 JVM 版本文件为参考，确定额外的字符集是否可用。）注意，尽管所有的 JVM 必须至少支持下列字符集，但是没有指定默认字符集，也没有要求是这些标准字符集之一。在 JVM 启动时确定默认值，取决于潜在的操作系统环境、区域设置、和/或 JVM 配置。如果您需要一个指定的字符集，最安全的办法是明确的命名它。不要假设默认部署与您的开发环境相同。

---

3 或者看起来是这样的。Unicode 现在可以定义大于 16 位的字符编码。直到 1.5 版本，Jave 才支持这些新的、扩展的编码。

表 6-1. 必要字符集	
字符集名称	**描述**
US-ASCII	7-位 ASCII，ISO 646-US。Unicode 字符集的基本拉丁模块。这是常见的美式英语字符集。
ISO-8859-1	ISO-LATIN-1。大部分欧洲语言使用的字符集。这是 US-ASCII 的扩展集并囊括了大部分非英语的欧洲字符。（见 http://www.unicode.org/charts/。）ISO-LATIN-1 的字符在 8 位内进行编码。
UTF-8	8-位 UCS 转换格式。由 RFC2279 以及 Unicode 标准 3.0（修正版）指定。这是字位导向的字符编码。小于 0x80 的 ASCII 字符被编码为单字节。其他字符被编码为两个或多个字节。对于多个序列，用首字节的高序位编码下面字节的数量。（见 http://www.ietf.org/rfc/rfc2279.txt。）UTF-8 与 ASCII 的互操作良好，因为简单的 ASCII 文件就是良好的 UTF-8 编码，而小于 0x80 的字符的 UTF-8 编码就是 ASCII 文件。
UTF-16BE	16-位 UCS 转换格式，高位字序。每个 Unicode 字符都被编码为 2-字节序列，首先编写高序位 8 位（bit）。
UTF-16LE	16-位 UCS 转换格式，低位字序。每个 Unicode 字符都被编码为 2-字节序列，首先编写低序位 8 位（bit）。
UTF-16	16-位 UCS 转换格式。字序由可选的字序标记确定。UTF-16 字符在 RFC2781 中指定。UTF-16BE 和 UTF-16BE 格式编码成 16-位，因而由字序确定。UTF-16 是可移植编码，它使用导向字节标记来表示编码字节流的其余部分是否是 UTF-16BE 或 UTF-16LE。见表 6-2。

字符集名称不区分大小写，也就是，当比较字符集名称时认为大写字母和小写字母相同。

互联网名称分配机构（IANA）维护所有正式注册的字符集名称，而表 6-1 中列出的所有名称都是在 IANA 注册的标准名称。

UTF-16BE 和 UTF-16LE 把每个字符编码为一个 2-字节数值。因此这类编码的解码器必须要预先了解数据是如何编码的，或者根据编码数据流本身来确定字节顺序的方式。UTF-16 编码承认一种字节顺序标记：Unicode 字符\uFEFF。只有发生在编码流的开端时字节顺序标记才表现为其特殊含义。如果之后遇到该值，它是根据其定义的 Unicode 值（零宽度，无间断空格）被映射。外来的，小字节序系统可能会优先考虑\uFEF 并且把流编码为 UTF-16LE。使用 UTF-16 编码优先考虑和认可字节顺序标记使系统带有不同的内部字节顺序，从而与 Unicode 数据交流。

表 6-2 解释了 Java 平台针对不同的组合所采取的操作。

表 6-2. UTF-16 字符集编码/解码			
	UTF-16	**UTF-16BE**	**UTF-16LE**
编码	预先考虑字节标记\uFEFF，编码为 UTF-16BE。	无字节标记，编码高位字序。	无字节标记，编码低位字序。
解码：无字节标记	解码为 UTF-16BE(Java 的原始字序)	解码，假设为高位字序。	解码，假设为低位字序。
解码：标记=\uFEFF	放弃标记。解码为 UTF-16BE.	放弃标记。解码为高位字序。	放弃标记。解码为低位字序。注意解码将交换字节，可能会引起运行异常。
解码：标记=\uFEFF	放弃标记。解码为 UTF-16LE.	放弃标记。解码为高位字序。注意解码将交换字节，可能会引起运行异常。	放弃标记。解码为低位字序。

示例 6-1 演示了通过不同的 *Charset* 实现如何把字符翻译成字节序列。

示例 6-1.使用标准字符集编码

```
package com.ronsoft.books.nio.charset;
import java.nio.charset.Charset;
import java.nio.ByteBuffer;
/**
 * Charset encoding test.  Run the same input string, which contains
 * some non-ascii characters, through several Charset encoders and dump out
 * the hex values of the resulting byte sequences.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class EncodeTest
{
    public static void main (String [] argv)
        throws Exception
    {
        // This is the character sequence to encode
        String input = "\u00bfMa\u00f1ana?";
        // the list of charsets to encode with
        String [] charsetNames = {
            "US-ASCII", "ISO-8859-1", "UTF-8", "UTF-16BE",
            "UTF-16LE", "UTF-16" // , "X-ROT13"
        };
        for (int i = 0; i < charsetNames.length; i++) {
            doEncode (Charset.forName (charsetNames [i]), input);
        }
    }
/**
     * For a given Charset and input string, encode the chars
     * and print out the resulting byte encoding in a readable form.
```

```
     */
    private static void doEncode (Charset cs, String input)
    {
       ByteBuffer bb = cs.encode (input);
       System.out.println ("Charset: " + cs.name(  ));
       System.out.println ("  Input: " + input);
       System.out.println ("Encoded: ");
       for (int i = 0; bb.hasRemaining(  ); i++) {
          int b = bb.get(  );
          int ival = ((int) b) & 0xff;
          char c = (char) ival;
          // Keep tabular alignment pretty
          if (i < 10) System.out.print (" ");
          // Print index number
          System.out.print ("  " + i + ": ");
          // Better formatted output is coming someday...
          if (ival < 16) System.out.print ("0");
          // Print the hex value of the byte
          System.out.print (Integer.toHexString (ival));
          // If the byte seems to be the value of a
          // printable character, print it.  No guarantee
          // it will be.
          if (Character.isWhitespace (c) ||
             Character.isISOControl (c))
          {
             System.out.println ("");
          } else {
             System.out.println (" (" + c + ")");
          }
       }
       System.out.println ("");
    }
}
```

下面是运行 EncodeTest 的输出结果：

```
Charset: US-ASCII
  Input: żMańana?
Encoded:
   0: 3f (?)
   1: 4d (M)
   2: 61 (a)
   3: 3f (?)
   4: 61 (a)
   5: 6e (n)
   6: 61 (a)
   7: 3f (?)
Charset: ISO-8859-1
  Input: żMańana?
Encoded:
   0: bf (ż)
   1: 4d (M)
   2: 61 (a)
   3: f1 (ń)
   4: 61 (a)
   5: 6e (n)
   6: 61 (a)
   7: 3f (?)
Charset: UTF-8
  Input: żMańana?
Encoded:
   0: c2 (Ż)
   1: bf (ż)
   2: 4d (M)
   3: 61 (a)
   4: c3 (Ă)
```

```
   5: b1 (±)
   6: 61 (a)
   7: 6e (n)
   8: 61 (a)
   9: 3f (?)
Charset: UTF-16BE
  Input: żMańana?
Encoded:
   0: 00
   1: bf (ż)
   2: 00
   3: 4d (M)
   4: 00
   5: 61 (a)
   6: 00
   7: f1 (ń)
   8: 00
   9: 61 (a)
  10: 00
  11: 6e (n)
  12: 00
  13: 61 (a)
  14: 00
  15: 3f (?)
Charset: UTF-16LE
  Input: żMańana?
Encoded:
   0: bf (ż)
   1: 00
   2: 4d (M)
   3: 00
   4: 61 (a)
   5: 00
   6: f1 (ń)
   7: 00
   8: 61 (a)
   9: 00
  10: 6e (n)
  11: 00
  12: 61 (a)
  13: 00
14: 3f (?)
  15: 00
Charset: UTF-16
  Input: żMańana?
Encoded:
   0: fe (ć)
   1: ff (˙)
   2: 00
   3: bf (ż)
   4: 00
   5: 4d (M)
   6: 00
   7: 61 (a)
   8: 00
   9: f1 (ń)
  10: 00
  11: 61 (a)
  12: 00
  13: 6e (n)
  14: 00
  15: 61 (a)
  16: 00

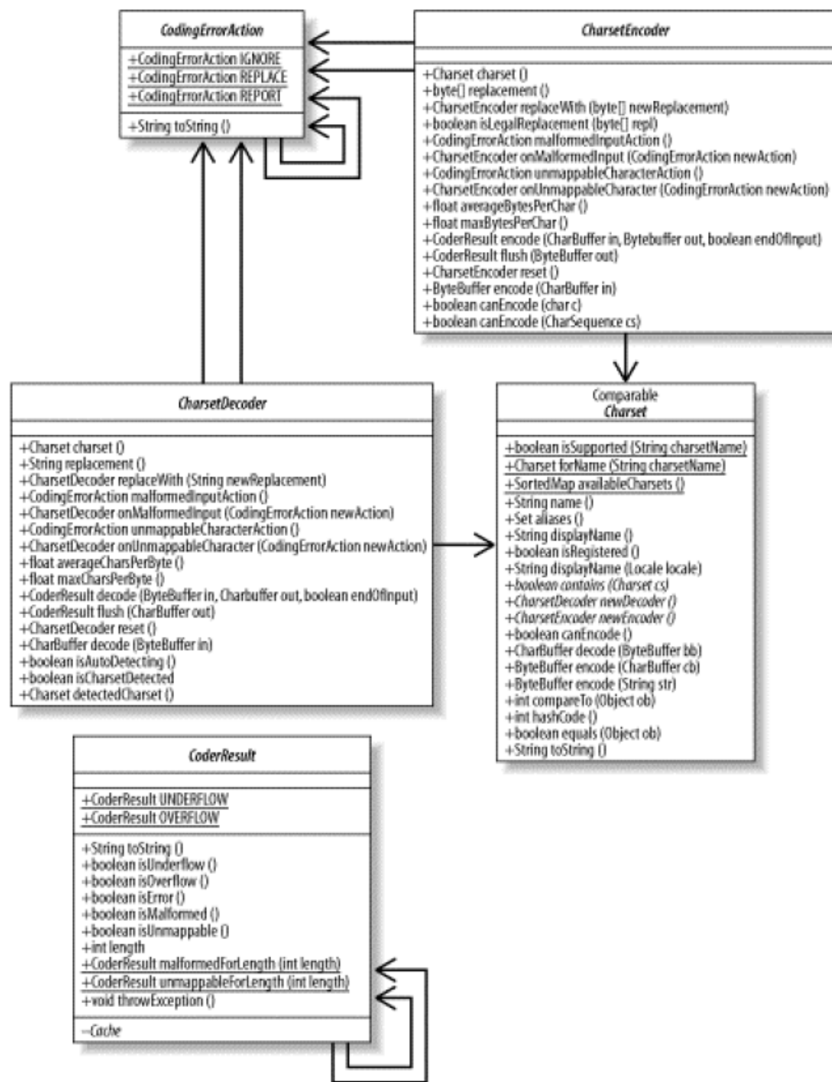
17: 3f (?)
```

6.2.1 字符集类

让我们深入到 *Charset* 类的 API 中（用图 6-2 概括说明）：

```
package java.nio.charset;
public abstract class Charset implements Comparable
{
       public static boolean isSupported (String charsetName)
       public static Charset forName (String charsetName)
       public static SortedMap availableCharsets(  )
       public final String name(  )
       public final Set aliases(  )
       public String displayName(  )
       public String displayName (Locale locale)
       public final boolean isRegistered(  )
       public boolean canEncode(  )
       public abstract CharsetEncoder newEncoder(  );
       public final ByteBuffer encode (CharBuffer cb)
       public final ByteBuffer encode (String str)
       public abstract CharsetDecoder newDecoder(  );
       public final CharBuffer decode (ByteBuffer bb)
       public abstract boolean contains (Charset cs);
       public final boolean equals (Object ob)
       public final int compareTo (Object ob)
       public final int hashCode(  )
       public final String toString(  )
}
```

图 6-2. 字符集类



Charset 类封装特定字符集的永恒信息。*Charset* 是抽取。通过调用静态工厂方法 *forName*()获得具体实例，导入所需字符集的名称。所有的 *Charset* 方法都是线程安全的；单一实例可以在多个线程中共享。

可以调用布尔类（boolean class）方法 *isSupported()*来确定在 JVM 运行中当前指定的字符集是否可用。通过 *Charset* SPI 机制可以动态安装新的字符集，所以给定字符集名称的答案可以随时间变化。在第 6.3 节中论述了 *Charset* SPI。

一个字符集可以有多个名称。通常它有一个规范名称但是也有零个或多个别名。规范名称或别名都可以通过 *forName()*和 *isSupported()*进行使用。

一些字符集也有历史遗留的名称，它们用于之前的 Java 平台版本并且向后兼容。字符

集的历史名称是由 *InputStreamReader* 和 *OutputStream-Writer* 类的 *getEncoding()*返回。如果字符集有历史名称，那么它将是规范名称或者 *Charset* 的别名之一。*Charset* 类不提供指出历史名称的标示。

静态类方法的最后一个，*availableCharsets()*，将返回在 JVM 中当前有效的所有字符集的 *java.util.SortedMap*。正如 *isSupported()*，如果安装新的字符集返回的值会随着时间改变。返回映射的成员将是用它们的规范名称作为密钥的 *Charset* 对象。迭代时，映射将根据规范名称按字母顺序排列。

*availableCharsets()*方法可能很慢。虽然支持许多字符集，但是通常它们只在有明确要求时才能被创建。调用 *availableCharsets()*需要实例化所有已知的 *Charset* 对象。实例化 *Charset* 可能需要加载库、网络资源访问、翻译表计算等等。如果您知道要使用的字符集的名称，使用 *forName()*方法。当您需要列举所有可用的字符集时使用 *availableCharsets()*—例如，面向交互用户展示一个选项。假设在此期间未安装新的字符集，通过 *availableCharsets()*返回的 *Map* 精确的包含了用 *forName()*可返回的相同的字符集。

一旦获取 *Charset* 实例的参数，*name()*方法将返回字符集的规范名称，并且 *aliases()*将给出包含别名的 *Set*。由 *aliases()*返回的 *Set* 将永远不会是 `null`，但是可以为空。

每个 *Charset* 对象也有两个 *displayName()*方法。这些方法的默认实现仅仅返回规范字符集名称。这些方法可以提供本地化的显示名称，例如，用在菜单或选项中。*displayName()*方法可以使用 *Locale* 参数指定一个地方化的环境。无参数版本使用默认区域设置。

如本节开端提到的，IANA 是维护字符集名称的权威登记机构。如果给出的 *Charset* 对象表示在 IANA 注册的字符集，那么 *isRegistered()*方法将返回 `true`。如果是这样的话，那么 *Charset* 对象需要满足几个条件：

- 字符集的规范名称应与在 IANA 注册的名称相符。
- 如果 IANA 用同一个字符集注册了多个名称，对象返回的规范名称应该与 IANA 注册中的 MIME-首选名称相符。
- 如果字符集名称从注册中移除，那么当前的规范名称应保留为别名。
- 如果字符集没有在 IANA 注册，它的规范名称必须以“X-”或“x-”开头。

大多数情况下，只有 JVM 卖家才会关注这些规则。然而，如果您打算以您自己的字符集作为应用的一部分，那么了解这些不该做的事情将对您很有帮助。针对 *isRegistered()*您应该返回 `false` 并以“X-”开头命名您的字符集。见 6.3 节。

6.2.2 字符集比较

下面的代码中包含了我们将在本节中详述的 *Charset* 的 API 方法：

```
public abstract class Charset implements Comparable
{
      // This is a partial API listing
      public abstract boolean contains (Charset cs);
      public final boolean equals (Object ob)
      public final int compareTo (Object ob)
      public final int hashCode(  )
      public final String toString(  )
}
```

回想一下，字符集是由字符的编码集与该字符集的编码方案组成的。与普通的集合类似，一个字符集可能是另一个字符集的子集。一个字符集（C_1）包含另一个（C_2），表示在 C_2 中表达的每个字符都可以在 C_1 中进行相同的表达。每个字符集都被认为是包含其本身。如果这个包含关系成立，那么您在 C_2（被包含的子集）中编码的任意流在 C_1 中也一定可以编码，无需任何替换。

*contains()*实例方法显示作为参数传入的 *Charset* 对象是否被该 *Charset* 对象封装的字符集所包含。该方法不能在运行时动态比较字符集；只有当具体的 *Charset* 类确定给出的字符集被包含的情况下才返回 `true`。如果 *contains()*返回 `false`，表示包含关系不存在或未知的包含关系。

如果一个字符集被另一个包含，这不意味着产生的编码字节序列将会等同于给定的输入字符序列。

Charset 类明确地覆盖了 *Object.equals()*方法。如果 *Charset* 的实例拥有相同的规范名称（由 *name()*返回），它们就被认为是相同的。在 JDK1.4.0 版本中，由 *equals()*实现的比较是规范名称串的简单比较，这意味着在测试过程中区分大小写。这是在未来的版本中应该更正的错误程序。由于 *Charset.equals()*方法覆盖了 *Object* 类中的默认方法，它必须声明为接受一个 *Object* 类的参数，而不是 *Charset* 类。*Charset* 类的对象永远不会与其他任意类的对象相等。



由于 JDK1.4 中 *Charset* 类的特殊实现方式，调用所有映射到同一字符集上的 *forName ()*方法，将返回相同的对象句柄。这意味着使用==运算符比较对象的引用的效果与使用 *equals()*方法的效果一样好。但请务必总是使用 *equals()*来检查两个对象是否相等。如果不这么做，将来这个类的实现更改时，您的代码就会失效。

您可能注意到了之前的代码清单中 *Charset* 实现了 *Comparable* 接口，表示它提供 *compareTo()* 方法。与 *equals()*类似的, *compareTo()*以 *Charset* 对象的规范名称为基础，返回结果。用 *Charset* 的 *compareTo()*方法进行比较时大小写不计。如果对 *Charset* 对象的集合进行排序，它们将按规范名称的顺序排列，大小写不计。再一次的，因为在 *Comparable* 中定义的 *compareTo()*方法采用 *Object* 作为参数类型，所以此处定义的这个也一样。如果您使用非 *Charset* 对象作为 *compareTo()*方法的参数，将产生 *ClassCastException*。*compareTo()*不能比较不同类型的对象实例。

继续用它们的规范名称鉴别 *Charset* 对象，*hashCode()*方法返回由 *name()*方法（意思是散列代码（也叫哈希代码）区分大小写）返回的 *String* 的散列代码。*Charset* 的 *toString()*方法返回规范名称。大部分时间，*hashCode()*的实现和 *toString()*方法不起什么作用。在这里提到它们是因为 *Charset* 类覆盖它们，如果在散列映射中使用可能会影响它们的表现或者在调试器中影响它们的出现方式。

现在我们已经学习了简单的 API 方法，让我们来看一下字符集编码器。这是字符和字节流之间实际完成转化的地方。

6.2.3 字符集编码器

字符集是由一个编码字符集和一个相关编码方案组成的。*CharsetEncoder* 和 *CharsetDecoder* 类实现转换方案。（见图 6-1.）

```
public abstract class Charset implements Comparable
{
      // This is a partial API listing
      public abstract boolean contains (Charset cs);
      public final boolean equals (Object ob)
      public final int compareTo (Object ob)
      public final int hashCode(  )
      public final String toString(  )
}
```

这里有用的第一个 API 方法是 *canEncode()*。该方法表示这个字符集是否允许编码。几乎所有的字符集都支持编码。主要的例外情况是带有解码器的字符集，它们可以自检测字节序列是如何编码并且之后会选择一个合适的解码方案。这些字符集通常只支持解码并且不创建自己的编码。

如果该 *Charset* 对象能够编码字符序列，*canEncode()*方法返回 `true`。如果为 `false`，上面列出的其他三个方法不应该在那个对象上被调用。这样做将引发 *UnsupportedOperationException*。

调用 *newEncoder()*返回 *CharsetEncoder* 对象，可以使用和字符集相关的编码方案把字符序列转化为字节序列。之后我们将在本节中学习 *CharsetEncoder* 类的 API，但是首先我们要快速浏览一下 *Charset* 余下的两个方法。

Charset 的两个 *encode()*方法使用方便，用默认值针对和字符集相关的编码器实现编码。两个都返回新 *ByteBuffer* 对象，包含符合给定的 *String* 或 *CharBuffer* 字符的一个编码字节序列。解码器通常都在 *CharBuffer* 对象上运行。*encode()*的形式采用 *String* 参数自动的为您创建一个临时的 *CharBuffer*，等同于下面这个：

```
charset.encode (CharBuffer.wrap (string));
```

在 *Charset* 对象上调用 *encode()*使用编码器的默认设置，等同于下列代码：

```
charset.newEncoder( )
      .onMalformedInput (CodingErrorAction.REPLACE)
      .onUnmappableCharacter (CodingErrorAction.REPLACE)
      .encode (charBuffer);
```

之后我们将进行讨论，它运行解码器，用默认字节序列替换任意的未识别或无效输入字符。

让我们针对 *CharsetEncoder* 浏览一下 API，从而更全面的理解字符编码的过程：

```
package java.nio.charset;
public abstract class CharsetEncoder
{
       public final Charset charset( )
       public final float averageBytesPerChar( )
       public final float maxBytesPerChar( )
       public final CharsetEncoder reset( )
       public final ByteBuffer encode (CharBuffer in) throws
             CharacterCodingException
       public final CoderResult encode (CharBuffer in, ByteBuffer out,
             boolean endOfInput)
       public final CoderResult flush (ByteBuffer out)
       public boolean canEncode (char c)
       public boolean canEncode (CharSequence cs)
       public CodingErrorAction malformedInputAction( )
       public    final   CharsetEncoder   onMalformedInput
                  (CodingErrorAction
newAction)
       public CodingErrorAction unmappableCharacterAction( )
       public final CharsetEncoder onUnmappableCharacter (
             CodingErrorAction newAction)
       public final byte [] replacement( )
       public boolean isLegalReplacement (byte[] repl)
       public final CharsetEncoder replaceWith (byte[] newReplacement)
}
```

CharsetEncoder 对象是一个状态转换引擎：字符进去，字节出来。一些编码器的调用可能需要完成转换。编码器存储在调用之间转换的状态。

这里列出的首个方法组提供跟 *CharsetEncoder* 对象有关的永恒信息。每个编码器和一个 *Charset* 对象相关联，而 *charset()*方法返回一个备份参考。

*averageBytesPerChar()*方法返回一个浮点值，表示编码集合的字符所需的平均字节数量。注意该值可以是分数值。当编码字符时，编码运算法则可以选择调节字节边界，或者一些字符可以编码成大于其他字节的字节（UTF-8 就是这样工作的）。该方法作为一个程序的探索很有用，用来确定 *ByteBuffer* 的近似尺寸，*ByteBuffer* 需要包含给定字符的编码字节。

最后，*maxBytesPerChar()*方法表示在集合中编码单字符需要的最大字节数。这也是一个浮点值。与 *averageBytesPerChar()*类似，该方法被用来按大小排列 *ByteBuffer*。用 *maxBytesPerChar()*返回的值乘以被编码的字符数量将得出最坏情况输出缓冲区大小。

在我们进入编码的要点之前，关于 *CharsetEncoder* API 的一个注意事项：首先，越简单的

*encode()*形式越方便，在重新分配的 *ByteBuffer* 中您提供的 *CharBuffer* 的编码集所有的编码于一身。这是当您在 *Charset* 类上直接调用 *encode()*时最后调用的方法。

当使用 *CharsetEncoder* 对象时，在编码之前或编码期间有设置错误处理参数的选项。（本节的后半段将讨论处理编码错误。）调用 *encode()*的单参数形式实现完整的编码循环（复位，编码以及清理），所以编码器之前的内状态将丢失。

让我们详细了解一些编码处理的工作原理。*CharsetEncoder* 类是一个状态编码引擎。实际上，编码器有状态意味着它们不是线程安全的：*CharsetEncoder* 对象不应该在线程中共享。编码可以在一个简单的步骤中完成，如上面提到的 *encode()*的首个形式，或者重复调用 *encode()*的第二个形式。编码过程如下：

1. 通过调用 *reset()*方法复位编码器的状态。让编码引擎准备开始产生编码字节流。新建的 *CharsetEncoder* 对象不需要复位，但是这么做也无妨。
2. 不调用或多次调用 *encode()*为编码器提供字符，`endOfnput` 参数 `false` 表示后面可能有更多的字符。给定的 *CharBuffer* 将消耗字符，而编码字节序列将被添加到提供的 *ByteBuffe*r 上。

   返回时，输入 *CharBuffer* 可能不是全部为空。可能填入输出 *ByteBuffer*，或者编码器可能需要更多的输入来完成多字符转化。编码器本身可能也保留可以影响序列转化实现的状态。在重新填入前紧凑输入缓冲区。

3. 最后一次调用 *encode()*，针对 `endOfInput` 参数导入 `true`。提供的 *CharBuffer* 可能包含额外的需要编码的字符或为空。重要的是 `endOfInput` 在最后的调用上为 `true`。这样就通知编码引擎后面没有输入了，允许它探测有缺陷的输入。
4. 调用 *flush()*方法来完成未完成的编码并输出所有剩下的字节。如果在输出 *ByteBuffer* 中没有足够的空间，需要多次调用该方法。

当消耗了所有的输入时，当输出 *ByteBuffer* 为满时，或者当探测到编码错误时，*encode()*方法返回。无论如何，将会返回 *CoderResult* 对象，来表示发生的情况。结果对象可表示下列结果条件之一：

Underflow（下溢）

正常情况，表示需要更多的输入。或者是输入 *CharBuffer* 内容不足；或者，如果它不为空，在没有额外的输入的情况下，余下的字符无法进行处理。更新 *CharBuffer* 的位置解决被编码器消耗的字符的问题。

在 *CharBuffer* 中填入更多的编码字符（首先在缓冲区上调用 *compact()*，如果是非空的情况）并再次调用 *encode()*继续。如果结束了，用空 *CharBuffer* 调用 *encode()*并且 `endOfInput` 为 `true`，之后调用 *flush()*确保所有的字节都被发送给 *ByteBuffer*。

下溢条件总是返回相同的对象实例：名为 CharsetEncoder.UNDERFLOW 的静态类变量。这就使您可以使用返回的对象句柄上的等号运算符（==）来对下溢进行检测。

Overflow（上溢）

表示编码器充满了输出 *ByteBuffer* 并且需要产生更多的编码输出。输入 *CharBuffer* 对象可

能会或可能不会被耗尽。这是正常条件，不表示出错。您应该消耗 *ByteBuffer* 但是不应该扰乱 *CharBuffer*，*CharBuffer* 将更新它的位置，之后再次调用 *encode()*。重复进行直到得到下溢结果。

与下溢类似的，上溢返回一致的实例，`CharsetEncoder.OVERFLOW`，它可直接用于等式比较。

Malformed input（有缺陷的输入）

编码时，这个通常意味着字符包含 16-位的数值，不是有效的 Unicode 字符。对于解码来说，这意味着解码器遭遇了不识别的字节序列。

返回的 *CoderResult* 实例将不是单一的参数，因为它是针对下溢和上溢的。见第 6.2.3.1 节 *CoderResult* 的 API。

Unmappable character（无映射字符）

表示编码器不能映射字符或字符的序列到字节上—例如，如果您正在使用 ISO-8859-1 编码但您的输入 *CharBuffer* 包含非-拉丁 Unicode 字符。对于解码，解码器知道输入字节序列但是不了解如何创建相符的字符。

编码时，如果编码器遭遇了有缺陷的或不能映射的输入，返回结果对象。您也可以检测独立的字符，或者字符序列，来确定它们是否能被编码。下面是检测能否进行编码的方法：

```
package java.nio.charset;

public abstract class CharsetEncoder
{
        // This is a partial API listing

        public boolean canEncode (char c)
        public boolean canEncode (CharSequence cs)
}
```

*canEncode()*的两个形式返回 `boolean` 结果，表示编码器是否能将给出的输入编码。两种方法都在一个临时的缓冲区内实现输入的编码。这将引起编码器内部状态的改变，所以当编码处理正在进行中时不应调用这些方法。开始编码处理前，使用这些方法检测您的输入。

*canEncode()*的第二个形成采用类型 *CharSequence* 的一个参数，在第五章介绍过。任何实现 *CharSequence*（当前 *CharBuffer, String,* 或 *StringBuffer*）的对象都可以导入到 *canEncode()*中。

CharsetEncoder 的剩下的方法包含在处理编码错误中：

```
public abstract class CharsetEncoder
{
       // This is a partial API listing
       public CodingErrorAction malformedInputAction(  )
       public    final  CharsetEncoder   onMalformedInput
                 (CodingErrorAction
newAction)
       public CodingErrorAction unmappableCharacterAction(  )
       public final CharsetEncoder onUnmappableCharacter (
```

```
                CodingErrorAction newAction)
        public final byte [] replacement(  )
        public boolean isLegalReplacement (byte[] repl)
        public final CharsetEncoder replaceWith (byte[] newReplacement)
}
```

如之前提到的，*CoderResult* 对象可以从 *encode()*中返回，表示编码字符序列的问题。有两个已定义的代码错误条件：*malformed* 和 *unmappable*。在每一个错误条件上都可以配置编码器实例来采取不同的操作。当这些条件之一发生时，*CodingErrorAction* 类封装可能采取的操作。*CodingErrorAction* 是无有用的方法的无价值的类。它是简单的，安全类型的列举，包含了它本身的静态、已命名的实例。*CodingErrorAction* 定义了三个公共域：

REPORT（报告）

创建 *CharsetEncoder* 时的默认行为。这个行为表示编码错误应该通过返回 *CoderResult* 对象报告，前面提到过。

IGNORE（忽略）

表示应忽略编码错误并且如果位置不对的话任何错误的输入都应中止。

REPLACE（替换）

通过中止错误的输入并输出针对该 *CharsetEncoder* 定义的当前的替换字节序列处理编码错误。

现在我们知道了可能的错误行为，如何使用之前的 API 清单中的前四个方法应该是相当明显的。*malformedInputAction()*方法返回针对有缺陷的输入生效的行为。调用 *onMalformedInput()*设置在那之后要使用的 *CodingErrorAction* 值。无法映射字符的一对类似方法设置错误行为，并返回 *CharsetEncode*r 对象句柄。通过返回 `CharsetEncoder`，这些方法允许调用链接。例如：

```
CharsetEncoder encoder = charset.newEncoder(  )
        .onMalformedInput (CodingErrorAction.REPLACE)
        .onUnmappableCharacter (CodingErrorAction.IGNORE);
```

当动作是 `CodingErrorAction.REPLACE` 时，在 *CharsetEncoder* 上的方法的最后一组处理要使用的替换序列的管理。

当前的替换字节序列可以通过调用 *replacement()*方法找回。如果您未设置自己的替换序列，将返回默认值。

您可以通过调用 *isLegalReplacement()*方法，用您想要使用的字节阵列检测替换序列的合法性。替换字节序列必须是对字符集有效的编码。记住，字符集编码把字符转化成字节序列，为以后的解码做准备。如果替换序列不能被解码成有效的字符序列，编码字节序列变为无效。

最后，您可以通过调用 *replaceWith()*设置新的替换序列并导入字节阵列。当编码错误发生时，将输出给定的字节序列并且相应的错误行为被设置为 `CodingErrorAction.REPLACE`。阵列中的字节序列必须是合法的替换值：如果不是，*java.lang.IllegalArgumentException* 将被抛出。返回值是 *CharsetEncoder* 对象本身。

6. 2. 3. 1 CoderResult 类

让我们看一下前面提到的 *CoderResult* 类。*CoderResult* 对象是由 *CharsetEncoder* 和 *CharsetDecoder* 对象返回的：

```
package java.nio.charset;
public class CoderResult {
      public static final CoderResult OVERFLOW
      public static final CoderResult UNDERFLOW
      public boolean isUnderflow(  )
      public boolean isOverflow(  )
public boolean isError(  )
      public boolean isMalformed(  )
      public boolean isUnmappable(  )
      public int length(  )
      public static CoderResult malformedForLength (int length)
      public static CoderResult unmappableForLength (int length)
public void throwException(  ) throws CharacterCodingException
}
```

如前面提到的，根据每个下溢和上溢条件返回 *CoderResult* 的一致实例。您可以看到在 *CoderResult* 类中，上述这些定义为公共静态域。这些实例可以使通用案例的检测变得更容易。您可以通过==运算符直接把 *CoderResult* 对象和这些公共域进行比较（见示例 6-2）。无论哪个 *CoderResult* 对象，您总是可以使用 API 来确定返回结果的含义。

前两个方法，*isUnderflow()*和 *isOverflow()*，不被认定是错误。如果其中一个方法返回 `true`，那么从 *CoderResult* 对象中不能再获取更多的信息。对于 *isUnderflow()*，*CoderResult.UNDERFLOW* 实例总是返回 `true`，对于 *isOverflow()*，*CoderResult.OVERFLOW* 总是返回 `true`。

其他两个布尔(boolean)函数，*isMalformed()*和 *isUnmappable()*，是错误条件。*isError()*方法是简便方法，如果其中一个方法返回 `true` 这个方法就返回 `true`。

如果 *CoderResult* 实例表示错误条件，*length()*方法通知您错误输入序列的长度。对于常规的下溢/上溢条件，没有相关的长度（就是为什么可以共享单实例）。如果您在 *CoderResult* 的实例上调用 *length()*，*CoderResult* 不表达错误（*isError()*返回 `false`）的情况下，它会抛出未检测的 *java.lang.UnsupportedOperation-Exception*，所以要小心。对于长度实例，输入 *CharBuffer* 会被放置在首个错误的字符上。

CoderResult 类同样也包含了三种简便方法，利于自定义编码器和解码器（6.3 节您将学习如何编写）的开发人员进行开发。*CoderResult* 构造函数是私有的：您不能直接对它进行实例化或者继承为自己的。我们已经看到 *CoderResult* 实例表示上溢和下溢是独立的。对于表示错误的实例，它们获取的唯一独特的位信息是 *length()*返回的值。两个工厂方法 *malformedForLength()*和 *unmappableForLength()*分别返回 *CoderResult* 的实例，从 *isMalformed()*或 *isUnmappable()*返回 `true`，并且它们的 *length()*方法返回您提供的值。这些工厂方法总是根据给出的长度返回相同的 *CoderResult* 实例。

在某些情况下，相比于沿着 *CoderResult* 对象导入，抛出一个异常要更合适一些。例如，

CharsetEncoder 类的全能 *encode()*方法，如果它遭遇编码错误则抛出一个异常。*throwException()*方法是简便方法，抛出 *CharsetCodingException* 适当的子类，详见表 6-3.

表 6-3.CoderResult.throwException()抛出的异常

结果类型	异常
isUnderflow()	*BufferUnderflowException*
isOverflow()	*BufferOverflowException*
isMalformed()	*MalformedInputException*
isUnmappable()	*UnmappableCharacterException*

6.2.4 字符集解码器

字符集解码器是编码器的逆转。通过特殊的编码方案把字节编码转化成 16-位 Unicode 字符的序列。与 *CharsetEncoder* 类似的, *CharsetDecoder* 是状态转换引擎。两个都不是线程安全的，因为调用它们的方法的同时也会改变它们的状态，并且这些状态会被保留下来。

```
package java.nio.charset;
public abstract class CharsetDecoder
{
        public final Charset charset(  )
        public final float averageCharsPerByte(  )
        public final float maxCharsPerByte(  )
        public boolean isAutoDetecting(  )
        public boolean isCharsetDetected(  )
        public Charset detectedCharset(  )
        public final CharsetDecoder reset(  )
        public final CharBuffer decode (ByteBuffer in)
              throws CharacterCodingException
        public final CoderResult decode (ByteBuffer in, CharBuffer out,
              boolean endOfInput)
        public final CoderResult flush (CharBuffer out)
        public CodingErrorAction malformedInputAction(  )
        public final CharsetDecoder onMalformedInput (
              CodingErrorAction newAction)
        public CodingErrorAction unmappableCharacterAction(  )
        public final CharsetDecoder onUnmappableCharacter (
              CodingErrorAction newAction)
       public final String replacement(  )
       public final CharsetDecoder replaceWith (String newReplacement)
}
```

像您能看到的，*CharsetDecoder* 的 API 几乎是 *CharsetEncoder* 的映像。本节我们将集中在差异上，继续进行您在 6.2.3 节中已经了解的假设。

前面清单中方法的第一个编组是不言自明的。通过调用 *charset()*可以获取相关的 *Charset* 对象。在这个编码中来自每个字节的字符解码的平均和最大数是分别通过 *averageCharsPerByte()*和 *maxCharsPerByte()*返回的。使用这些值可以为 *CharBuffer* 对象排序，来接收解码的字符。

CharsetDecoder 类有其独特的方法集。在前面的清单中，这些方法不得不处理字符集自检测。首个方法，*isAutoDetecting()*，返回表示这个解码器是否能够自检测编码字节序列使用的编码方法的布尔值。

如果 *isAutoDetecting()*返回 `true`，那么前面清单后面的两个方法是有意义的。如果解码器能够从输入字节序列中读取足够的字节来确定使用的编码类型，那么 *isCharsetDetected()*方法将返回 `true`。只有当解码程序已经开始时这个方法才有作用（因为它必须读取一些字节并进行检测）。调用 *reset()*之后，它将总是返回 `false`。这个方法可选并且只对自检测字符集有意义。默认实现总是抛出 *java.lang.UnsupportedOperationException*。

如果字符集被检测（用 *isCharsetDetected()*返回 `true` 表示），那么通过调用 *detectedCharset()*可以获取表示那个字符集的 *Charset* 对象。除非您知道字符集已经被探测，否则不应该调用这个方法。如果解码器还没有读取足够的输入，不能确定用编码表达的字符集，将抛出 *java.lang.IllegalStateException*。*detectedCharset()*方法同样的也是可选，并且如果字符集没有被自探测，将抛出同一个 *java.lang.UnsupportedOperationException*。适当的使用 *isAutoDetecting()*和 *isCharsetDetected()*，您应该不会有什么问题。

现在，让我们转到实际完成解码的方法上：

```
package java.nio.charset;
public abstract class CharsetDecoder
{
      // This is a partial API listing
      public final CharsetDecoder reset(  )
      public final CharBuffer decode (ByteBuffer in)
            throws CharacterCodingException
      public final CoderResult decode (ByteBuffer in, CharBuffer out,
            boolean endOfInput)
      public final CoderResult flush (CharBuffer out)
}
```

解码处理和编码类似，包含相同的基本步骤：

1. 复位解码器，通过调用 *reset()*，把解码器放在一个已知的状态准备用来接收输入。
2. 把 *endOfInput* 设置成 `false` 不调用或多次调用 *decode()*，供给字节到解码引擎中。随着解码的进行，字符将被添加到给定的 *CharBuffer* 中。
3. 把 `endOfInput` 设置成 `true` 调用一次 *decode()*，通知解码器已经提供了所有的输入。
4. 调用 *flush()*，确保所有的解码字符都已经发送给输出。

这在本质上和编码相同（详见 6.2.3 节）。*decode()*方法同样的也返回 *CoderResult* 对象，表示发生的情况。这些结果对象的意义和 *CharsetEncoder.encode()*返回的意义相同。

当返回下溢或上溢指示时，输入和输出缓冲区应该以跟编码相同的方式进行管理。

现在，处理错误的方法：

```
package java.nio.charset;
public abstract class CharsetDecoder
{
      // This is a partial API listing
      public CodingErrorAction malformedInputAction(  )
      public final CharsetDecoder onMalformedInput (
            CodingErrorAction newAction)
      public CodingErrorAction unmappableCharacterAction(  )
      public final CharsetDecoder onUnmappableCharacter (
            CodingErrorAction newAction)
```

```
        public final String replacement(  )
        public final CharsetDecoder replaceWith (String newReplacement)
}
```

API 方法处理在 *String* 上运行的替换序列，而不是在字节阵列上。解码时，字节序列被转化成字符序列，所以解码操作的替换序列被指定为 *String*，*String* 包含要在错误条件上输出 *CharBuffer* 中要插入的字符。注意没有用来检测替换序列的 *isLegalReplacement()*方法。您构造的任何字符串都是合法的替换序列，除非它不再是 *maxCharsPerByte()*返回的值。用字符串调用 *replaceWith()*，太长将导致 *java.lang.IllegalArgumentException*。

本节是有意地简洁扼要。详细信息，参考 6.2.3 节。

示例 6-2 说明了如何对表示字符集编码的字节流进行编码。

示例 6-2. 字符集解码

```
package com.ronsoft.books.nio.charset;
import java.nio.*;
import java.nio.charset.*;
import java.nio.channels.*;
import java.io.*;
/**
 * Test charset decoding.
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class CharsetDecode
{
    /**
     * Test charset decoding in the general case, detecting and handling
     * buffer under/overflow and flushing the decoder state at end of
     * input.
     * This code reads from stdin and decodes the ASCII-encoded byte
     * stream to chars.  The decoded chars are written to stdout.  This
* is effectively a 'cat' for input ascii files, but another charset
     * encoding could be used by simply specifying it on the command line.
     */
    public static void main (String [] argv)
        throws IOException
    {
        // Default charset is standard ASCII
        String charsetName = "ISO-8859-1";
        // Charset name can be specified on the command line
        if (argv.length > 0) {
            charsetName = argv [0];
        }
        // Wrap a Channel around stdin, wrap a channel around stdout,
        // find the named Charset and pass them to the decode method.
        // If the named charset is not valid, an exception of type
        // UnsupportedCharsetException will be thrown.
        decodeChannel (Channels.newChannel (System.in),
            new OutputStreamWriter (System.out),
            Charset.forName (charsetName));
    }
    /**
     * General purpose static method which reads bytes from a Channel,
     * decodes them according
     * @param source A ReadableByteChannel object which will be read to
     *  EOF as a source of encoded bytes.
     * @param writer A Writer object to which decoded chars will be
     *  written.
     * @param charset A Charset object, whose CharsetDecoder will be used
     *  to do the character set decoding.
```

```
     */
    public static void decodeChannel (ReadableByteChannel source,
       Writer writer, Charset charset)
       throws UnsupportedCharsetException, IOException
    {
       // Get a decoder instance from the Charset
       CharsetDecoder decoder = charset.newDecoder(  );
       // Tell decoder to replace bad chars with default mark
       decoder.onMalformedInput (CodingErrorAction.REPLACE);
       decoder.onUnmappableCharacter (CodingErrorAction.REPLACE);
       // Allocate radically different input and output buffer sizes
       // for testing purposes
       ByteBuffer bb = ByteBuffer.allocateDirect (16 * 1024);
       CharBuffer cb = CharBuffer.allocate (57);
       // Buffer starts empty; indicate input is needed
       CoderResult result = CoderResult.UNDERFLOW;
       boolean eof = false;
       while ( ! eof) {
          // Input buffer underflow; decoder wants more input
          if (result == CoderResult.UNDERFLOW) {
             // decoder consumed all input, prepare to refill
             bb.clear(  );
             // Fill the input buffer; watch for EOF
             eof = (source.read (bb) == -1);
             // Prepare the buffer for reading by decoder
             bb.flip(  );
          }
          // Decode input bytes to output chars; pass EOF flag
          result = decoder.decode (bb, cb, eof);
          // If output buffer is full, drain output
          if (result == CoderResult.OVERFLOW) {
             drainCharBuf (cb, writer);
          }
       }
       // Flush any remaining state from the decoder, being careful
       // to detect output buffer overflow(s)
       while (decoder.flush (cb) == CoderResult.OVERFLOW) {
          drainCharBuf (cb, writer);
       }
       // Drain any chars remaining in the output buffer
       drainCharBuf (cb, writer);
       // Close the channel; push out any buffered data to stdout
       source.close(  );
       writer.flush(  );
    }

    /**
     * Helper method to drain the char buffer and write its content to
     * the given Writer object.  Upon return, the buffer is empty and
     * ready to be refilled.
     * @param cb A CharBuffer containing chars to be written.
     * @param writer A Writer object to consume the chars in cb.
     */
    static void drainCharBuf (CharBuffer cb, Writer writer)
       throws IOException
    {
       cb.flip(  );       // Prepare buffer for draining
       // This writes the chars contained in the CharBuffer but
       // doesn't actually modify the state of the buffer.
       // If the char buffer was being drained by calls to get(  ),
       // a loop might be needed here.
       if (cb.hasRemaining(  )) {
          writer.write (cb.toString(  ));
       }
       cb.clear(  );       // Prepare buffer to be filled again
    }
}
```

字符集和相关的编码器与解码器差不多就是这些内容了。下一节将介绍 *Charset* SPI。

6.3 字符集服务器供应者接口

Charset SPI 为开发人员提供了一种机制，可以在运行中的 JAM 环境中添加新的 *Charset* 实现。如果您需要创建自己的字符集，或者导入您正在使用而 JVM 平台没有提供的字符集，那么您需要字符集 SPI。

可插拔的 SPI 结构是在许多不同的内容中贯穿于 Java 环境使用的。在 1.4JDK 中有八个包，一个叫 `spi` 而剩下的有其它的名称。可插拔是一个功能强大的设计技术，是在 Java 的可移植性和适应性上建立的基石之一。

字符集是由 IANA 正式定义，而标准的字符集是在那里注册的。从 1.4 起在 Java 中处理字符集是以 IANA 发布的惯例和标准为基础。IANA 不仅仅注册名称，而且对这些名称(RFC2278)的结构和内容也有规定。如果您创建新的 *Charset* 实现，您应该遵循字符集名称的惯例。*Charset* 类强迫实现相同的规则。例如，字符集的名称必须由表 6-4 中列出的字符集合组成，并且首字符必须是字母或阿拉伯数字。

表格 6-4. 字符集名称的合法字符

字符	Unicode 值	RFC2278 名称
A-Z	\u0041-\u005a	
a-z	\u0061-\u007a	
0-9	\u0030-\u0039	
-（破折号）	\u002d	HYPHEN-MINUS
.（句点）	\u002e	FULLSTOP
:（冒号）	\u003a	COLON
_（下划线）	\u005f	LOWLINE

在浏览 API 之前，需要解释一下 *Charset* SPI 如何工作。*java.nio.charset.spi* 包仅包含一个抽取类，*CharsetProvider*。这个类的具体实现供给与它们提供过的 *Charset* 对象相关的信息。为了定义自定义字符集，您首先必须从 `java.nio.charset package` 中创建 *Charset, CharsetEncoder*，以及 *CharsetDecoder* 的具体实现。然后您创建 *CharsetProvider* 的自定义子类，它将把那些类提供给 JVM。

6.3.2 节中列出了自定义字符集和提供方的完整取样实现。

6.3.1 创建自定义字符集

在了解 `java.nio.charset.spi` 包中的唯一的类之前，让我们在 `java.nio.charset` 中再花点时间并讨论一下实现自定义字符集需要什么。在您能使 *Charset* 对象在运行的 JVM 中有效之前，您需要创建它。让我们再看一下 *Charset* API，添加构造函数并为抽取方法添加注释：

```
package java.nio.charset;
public abstract class Charset implements Comparable
{
      protected Charset (String canonicalName, String [] aliases)
```

```
        public static SortedMap availableCharsets( )
        public static boolean isSupported (String charsetName)
        public static Charset forName (String charsetName)
        public final String name( )
        public final Set aliases( )
        public String displayName( )
public String displayName (Locale locale)
        public final boolean isRegistered( )
        public boolean canEncode( )
        public abstract CharsetEncoder newEncoder( );
        public final ByteBuffer encode (CharBuffer cb)
        public final ByteBuffer encode (String str)
        public abstract CharsetDecoder newDecoder( );
        public final CharBuffer decode (ByteBuffer bb)
        public abstract boolean contains (Charset cs);
        public final boolean equals (Object ob)
        public final int compareTo (Object ob)
        public final int hashCode( )
        public final String toString( )
}
```

您至少要做的是创建 *java.nio.charset.Charset* 的子类、提供三个抽取方法的具体实现以及一个构造函数。*Charset* 类没有默认的，无参数的构造函数。这表示您的自定义字符集类必须有一个构造函数，即使它不接受参数。这是因为您必须在实例化时调用 *Charset* 的构造函数（通过在您的构造函数的开端调用 *super()*），从而通过您的字符集规范名称和别名供给它。这样做可以让 *Charset* 类中的方法帮您处理和名称相关的事情，所以是件好事。

通过获取您自定义的编码器和解码器，三个抽取方法中的两个可以被工厂化。同样的您也要实现布尔方法 *contains()*，但是您可以用总是返回 `false` 提出它，`false` 表示您不知道您的字符集是否包含给出的字符集。大多数情况下，所有其他的 *Charset* 方法都有可以使用的默认实现。如果您的字符集有特殊需求，相应的覆盖默认方法。

同样地，您需要提供 *CharsetEncoder* 和 *CharsetDecoder* 的具体实现。回想一下，字符集是编码的字符和编码/解码方案的集合。如我们之前所看到的，编码和解码在 API 水平上几乎是对称的。这里给出了关于实现编码器所需要的东西的简短讨论：一样适用于建立解码器。这是 Char*setEncoder* 类的清单，用它的构造函数和受保护的以及添加的抽取方法：

```
ackage java.nio.charset;
public abstract class CharsetEncoder
{
        protected CharsetEncoder (Charset cs,
               float averageBytesPerChar, float maxBytesPerChar)
        protected CharsetEncoder (Charset cs,
               float averageBytesPerChar, float maxBytesPerChar,
               byte [] replacement)
        public final Charset charset( )
        public final float averageBytesPerChar( )
        public final float maxBytesPerChar( )
        public final CharsetEncoder reset( )
        protected void implReset( )
        public final ByteBuffer encode (CharBuffer in)
              throws CharacterCodingException
        public final CoderResult encode (CharBuffer in, ByteBuffer out,
              boolean endOfInput)
        public final CoderResult flush (ByteBuffer out)
        protected CoderResult implFlush(ByteBuffer out)
        public boolean canEncode (char c)
        public boolean canEncode (CharSequence cs)
```

```
        public CodingErrorAction malformedInputAction(  )
        public final CharsetEncoder onMalformedInput (
              CodingErrorAction newAction)
        protected void implOnMalformedInput (CodingErrorAction newAction)
        public CodingErrorAction unmappableCharacterAction(  )
        public final CharsetEncoder onUnmappableCharacter (
              CodingErrorAction newAction)
        protected void implOnUnmappableCharacter (
                 CodingErrorAction newAction)
        public final byte [] replacement(  )
        public boolean isLegalReplacement (byte[] repl)
        public final CharsetEncoder replaceWith (byte[] newReplacement)
        protected void implReplaceWith (byte[] newReplacement)
        protected abstract CoderResult encodeLoop (CharBuffer in,
               ByteBuffer out);
}
```

与 *Charset* 类似的, *CharsetEncoder* 没有默认的构造函数，所以您需要在具体类构造函数中调用 *super()*，提供需要的参数。

首先看一下最后一个方法。为了供给您自己的 *CharsetEncoder* 实现，您至少要提供具体 *encodeLoop()*方法。对于简单的编码运算法则，其他方法的默认实现应该可以正常进行。注意 *encodeLoop()*采用和 *encode()*的参数类似的参数，不包括布尔标志。*encode()*方法代表到 *encodeLoop()*的实际编码，它仅需要关注来自 *CharBuffer* 参数消耗的字符，并且输出编码的字节到提供的 *ByteBuffer* 上。

主要的 *encode()*方法负责存储调用和处理编码错误上的状态。与 *encode()*类似的，*encodeLoop()* 方法返回 *CoderResult* 对象，表示在处理缓冲区时发生的情况。如果您的 *encodeLoop()*填充了输出 *ByteByffer*，它应返回 `CoderResult.OVERFLOW`。如果输入 *CharBuffer* 耗尽了，应返回 `CoderResult.UNDERFLOW`。如果您的编码器需要比输入缓冲区中多的输出才能做出编码决定，您可以通过返回 `UNDERFLOW` 提前实现直到 *CharBuffer* 中有充足的输入继续。

上面所列剩下的受保护方法—以 `impl` 开始—是状态改变回调，当编码器的状态发生改变时通知实现（您的代码）。所有这些方法的默认实现是存根，不起作用。例如，如果在您的编码器中还有额外的状态，当编码器重置时您可能要知道。您不能覆盖 *reset()*方法本身，因为它声明为最后。当在 *CharsetEncoder* 上调用 *reset()*时为您提供 *implReset()*方法，让您以一种清晰的方式了解发生的情况。其他 `impl` 类针对感兴趣的其他事件扮演相同的角色。

作为参考，这是针对 *CharsetDecoder* 列出的相等的 API：

```
package java.nio.charset;
public abstract class CharsetDecoder
{
        protected CharsetDecoder (Charset cs, float averageCharsPerByte,
              float maxCharsPerByte)
        public final Charset charset(  )
        public final float averageCharsPerByte(  )
        public final float maxCharsPerByte(  )
        public boolean isAutoDetecting(  )
        public boolean isCharsetDetected(  )
        public Charset detectedCharset(  )
        public final CharsetDecoder reset(  )
        protected void implReset(  )
        public final CharBuffer decode (ByteBuffer in)
              throws CharacterCodingException
```

```
        public final CoderResult decode (ByteBuffer in, CharBuffer out,
              boolean endOfInput)
        public final CoderResult flush (CharBuffer out)
        protected CoderResult implFlush (CharBuffer out)
        public CodingErrorAction malformedInputAction(  )
        public final CharsetDecoder onMalformedInput (
              CodingErrorAction newAction)
        protected void implOnMalformedInput (CodingErrorAction newAction)
        public CodingErrorAction unmappableCharacterAction(  )
        public final CharsetDecoder onUnmappableCharacter (
              CodingErrorAction newAction)
        protected void implOnUnmappableCharacter (
                 CodingErrorAction newAction)
        public final String replacement(  )
        public final CharsetDecoder replaceWith (String newReplacement)
         protected void implReplaceWith (String newReplacement)
        protected abstract CoderResult decodeLoop (ByteBuffer in,
                 CharBuffer out);
}
```

现在，我们已经看到了如何实现自定义字符集，包括相关的编码器和解码器，让我们看一下如何把它们连接到 JVM 中，这样可以利用它们运行代码。

6.3.2 供给您的自定义字符集

为了给 JVM 运行时环境提供您自己的 *Charset* 实现，您必须在 `java.nio.charsets.-spi` 中创建 *CharsetProvider* 类的具体子类，每个都带有一个无参数构造函数。无参数构造函数很重要，因为您的 *CharsetProvider* 类将要通过读取配置文件的全部合格名称进行定位。之后这个类名称字符串将被导入到 *Class.newInstance()*来实例化您的提供方，它仅通过无参数构造函数起作用。

JVM 读取的配置文件定位字符集提供方，被命名为 *java.nio.charset.spi.CharsetProvider*。它在 JVM 类路径中位于源目录（META-INF/services）中。每一个 Java Archive（Java 档案文件）（JAR）都有一个 *META-INF* 目录，它可以包含在那个 JAR 中的类和资源的信息。一个名为 *META-INF* 的目录也可以在 JVM 类路径中放置在常规目录的顶端。

META-INF/services 目录中的每个文件都有一个完全合格的服务提供方类的名称。每个文件的内容都是完全合格类名称的清单，名称是那个类（所以文件内的每个类的命名都必须是用文件名称表达的类的 `instanceof`）的具体实现。JAR 的详细信息请查阅 http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html。

当用类加载器优先检查类路径组件时（JAR 或目录），如果存在 *META-INF/services* 目录，那么它包含的每个文件都将被处理。每个都被读取，并且列出的所有类都被实例化且注册为通过文件名称鉴别的类的提供方。在名为 *java.nio.charset.spi.CharsetProvider* 的文件中放置您的 *CharsetProvider* 的完全合格名称，您就是正在把它注册为字符集的提供方。

配置文件的格式是一个简单的完全合格的类名称的清单，每行一个。注释字符是#（#，`\u0023`）。文件必须在 UTF-8 内编码（标准文本文件）。这个服务清单内命名的类不用放在同一个 JAR 上，但是类必须对同一个背景的类加载器可见（也就是，处于同一个类路径）。如果同一个 *CharsetProvider* 类在不止一个服务文件上命名，或者在同一个文件上多次命名，那么它只将作

为服务提供方被添加一次。

这个机制易于安装新的 *CharsetProvider* 和它提供的 *Charset* 实现。JAR 包含您的字符集实现，以及服务文件命名，仅需要处于 JVM 的类路径中。您也可以在您的 JVM 中把它安装成一个扩展，在您的操作系统（大部分情况中是 *jre/lib/ext*）定义的扩展目录中放置 JAR 即可。然后您的自定义字符集在每次 JVM 运行时就可用了。

没有指定的 API 机制，把新的字符集添加到 JVM 编程中。独立的 JVM 实现可以提供 API，但是 JDK1.4 没有提供这样的方法。

现在我们知道如何使用 *CharsetProvider* 类添加字符集，让我们看一下代码。*CharsetProvider* 的 API 几乎是没有作用的。提供自定义字符集的实际工作是发生在创建自定义 *Charset，CharsetEncoder*，以及 *CharsetDecoder* 类中。*CharsetProvider* 仅是连接您的字符集和运行时环境的促进者。

```
package java.nio.charset.spi;
public abstract class CharsetProvider
{
      protected CharsetProvider(  ) throws SecurityException
      public abstract Iterator charsets(  );
      public abstract Charset charsetForName (String charsetName);
}
```

注意受保护的构造函数。*CharsetProvider* 不应该用您的代码直接实例化。*CharsetProvider* 对象将通过低水平的服务提供方设备被实例化。如果您需要建立提供方可用的字符集，在 `CharsetProvider` 类中定义默认的构造函数。这可以从外部资源中调用加载的字符集映射信息，通过算法产生转化的映射等等。同样的，注意 *CharsetProvider* 的构造函数可以抛出 *java.lang.SecurityException*。

CharsetProvider 对象的实例化由 *SecurityManager* 检查（如果安装的情况下）。安全管理必须允许 *java.lang.RuntionPermission*(“`charsetProvider`”)，或者不安装新的字符集提供方。字符集可以包含在安全敏感操作中，例如编码 URL 以及其他数据内容。潜在伤害是显著的。您可能需要安装一个安全管理，如果在您的应用中有不信任的正在运行的代码，阻止新的字符集。您也需要检查不信任的 JAR，用来看它们在 *META-INF/service* 下是否包含服务配置文件，用于安装自定义字符集提供方（或者任何种类的自定义服务提供方）。

用您提供的 *Charset* 实现的使用方调用 *CharsetProvider* 上定义的两个方法。大部分情况是，您的提供方将被 *Charset* 类的静态方法调用，用来发现可用字符集相关的信息，但是其他的类也可以调用这些方法。

调用 *Charsets()*方法来获取使您的提供方类可用的 *Charset* 类的清单。它应该返回 *java.util.Iterator*，列举到给出的 *Charset* 实例的参数上。*Charset.availableCharsets()*方法返回的映射是一个集合体，在每个当前安装的 *CharsetProvider* 实例上调用 *charsets()*方法的集合体。

另一种方法，*charsetForName()*，被调用用来映射字符集名称，规范名称或者别名，到 *Charset* 对象上。如果您的提供方没有通过要求的名称提供字符集，这个方法应返回 `null`。

这就是全部了。现在您有创建自己的自定义字符集所有的必要工具以及和它们相关的编码器和解码器，插入并激活它们，运行 JVM。示例 6-3 中演示了自定义 *Charset* 和 *CharsetProvider* 的实现，包含说明字符集使用的取样代码，编码和解码，以及 *Charset* SPI。示例 6-3 实现了一个自定义 *Charset*。

示例 6-3.自定义 Rot13 字符集

```
package com.ronsoft.books.nio.charset;
import java.nio.CharBuffer;
import java.nio.ByteBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetEncoder;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CoderResult;
import java.util.Map;
import java.util.Iterator;
import java.io.Writer;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.io.OutputStreamWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.FileReader;
/**
* A Charset implementation which performs Rot13 encoding.  Rot-13
encoding
 * is a simple text obfuscation algorithm which shifts alphabetical
 * characters by 13 so that 'a' becomes 'n', 'o' becomes 'b', etc.  This
 * algorithm was popularized by the Usenet discussion forums many years
ago
* to mask naughty words, hide answers to questions, and so on.  The Rot13
 * algorithm is symmetrical, applying it to text that has been scrambled
by
 * Rot13 will give you the original unscrambled text.
 *
 * Applying this Charset encoding to an output stream will cause
everything
 * you write to that stream to be Rot13 scrambled as it's written out.
And
 * appying it to an input stream causes data read to be Rot13 descrambled
 * as it's read.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class Rot13Charset extends Charset
{
    // the name of the base charset encoding we delegate to
    private static final String BASE_CHARSET_NAME = "UTF-8";
    // Handle to the real charset we'll use for transcoding between
    // characters and bytes.  Doing this allows us to apply the Rot13
    // algorithm to multibyte charset encodings. But only the
    // ASCII alpha chars will be rotated, regardless of the base encoding.
    Charset baseCharset;
    /**
     * Constructor for the Rot13 charset.  Call the superclass
     * constructor to pass along the name(s) we'll be known by.
     * Then save a reference to the delegate Charset.
     */
    protected Rot13Charset (String canonical, String [] aliases)
    {
        super (canonical, aliases);
        // Save the base charset we're delegating to
        baseCharset = Charset.forName (BASE_CHARSET_NAME);
```

```
    }
    // --------------------------------------------------------
/**
* Called by users of this Charset to obtain an encoder.
    * This implementation instantiates an instance of a private class
     * (defined below) and passes it an encoder from the base Charset.
     */
    public CharsetEncoder newEncoder(  )
    {
       return new Rot13Encoder (this, baseCharset.newEncoder(  ));
    }
    /**
     * Called by users of this Charset to obtain a decoder.
     * This implementation instantiates an instance of a private class
     * (defined below) and passes it a decoder from the base Charset.
     */
    public CharsetDecoder newDecoder(  )
    {
       return new Rot13Decoder (this, baseCharset.newDecoder(  ));
    }
    /**
     * This method must be implemented by concrete Charsets.  We always
     * say no, which is safe.
     */
    public boolean contains (Charset cs)
    {
       return (false);
    }
    /**
     * Common routine to rotate all the ASCII alpha chars in the given
     * CharBuffer by 13.  Note that this code explicitly compares for
     * upper and lower case ASCII chars rather than using the methods
     * Character.isLowerCase and Character.isUpperCase.  This is because
     * the rotate-by-13 scheme only works properly for the alphabetic
     * characters of the ASCII charset and those methods can return
     * true for non-ASCII Unicode chars.
     */
    private void rot13 (CharBuffer cb)
    {
       for (int pos = cb.position(); pos < cb.limit(  ); pos++) {
          char c = cb.get (pos);
          char a = '\u0000';
          // Is it lowercase alpha?
          if ((c >= 'a') && (c <= 'z')) {
             a = 'a';
          }
          // Is it uppercase alpha?
          if ((c >= 'A') && (c <= 'Z')) {
             a = 'A';
          }
          // If either, roll it by 13
          if (a != '\u0000') {
             c = (char)((((c - a) + 13) % 26) + a);
             cb.put (pos, c);
          }
       }
    }
// --------------------------------------------------------
    /**
    * The encoder implementation for the Rot13 Charset.
     * This class, and the matching decoder class below, should also
     * override the "impl" methods, such as implOnMalformedInput(  ) and
     * make passthrough calls to the baseEncoder object.  That is left
     * as an exercise for the hacker.
     */
    private class Rot13Encoder extends CharsetEncoder
    {
       private CharsetEncoder baseEncoder;
```

```
        /**
         * Constructor, call the superclass constructor with the
         * Charset object and the encodings sizes from the
         * delegate encoder.
         */
        Rot13Encoder (Charset cs, CharsetEncoder baseEncoder)
        {
           super (cs, baseEncoder.averageBytesPerChar(  ),
              baseEncoder.maxBytesPerChar(  ));
           this.baseEncoder = baseEncoder;
        }
        /**
         * Implementation of the encoding loop.  First, we apply
         * the Rot13 scrambling algorithm to the CharBuffer, then
         * reset the encoder for the base Charset and call it's
         * encode(  ) method to do the actual encoding.  This may not
         * work properly for non-Latin charsets.  The CharBuffer
         * passed in may be read-only or re-used by the caller for
         * other purposes so we duplicate it and apply the Rot13
         * encoding to the copy.  We DO want to advance the position
         * of the input buffer to reflect the chars consumed.
         */
        protected CoderResult encodeLoop (CharBuffer cb, ByteBuffer bb)
        {
           CharBuffer tmpcb = CharBuffer.allocate (cb.remaining(  ));
           while (cb.hasRemaining(  )) {
              tmpcb.put (cb.get(  ));
           }
           tmpcb.rewind(  );
           rot13 (tmpcb);
           baseEncoder.reset(  );
           CoderResult cr = baseEncoder.encode (tmpcb, bb, true);
           // If error or output overflow, we need to adjust
           // the position of the input buffer to match what
           // was really consumed from the temp buffer.  If
           // underflow (all input consumed), this is a no-op.
           cb.position (cb.position() - tmpcb.remaining(  ));
           return (cr);
        }
     }
  // -----------------------------------------------------------
     /**
      * The decoder implementation for the Rot13 Charset.
      */
     private class Rot13Decoder extends CharsetDecoder
     {
        private CharsetDecoder baseDecoder;
        /**
         * Constructor, call the superclass constructor with the
         * Charset object and pass alon the chars/byte values
         * from the delegate decoder.
         */
        Rot13Decoder (Charset cs, CharsetDecoder baseDecoder)
        {
           super (cs, baseDecoder.averageCharsPerByte(  ),
              baseDecoder.maxCharsPerByte(  ));
           this.baseDecoder = baseDecoder;
        }
        /**
         * Implementation of the decoding loop.  First, we reset
         * the decoder for the base charset, then call it to decode
         * the bytes into characters, saving the result code.  The
         * CharBuffer is then de-scrambled with the Rot13 algorithm
         * and the result code is returned.  This may not
         * work properly for non-Latin charsets.
         */
       protected CoderResult decodeLoop (ByteBuffer bb, CharBuffer cb)
        {
```

```
                baseDecoder.reset( );
                CoderResult result = baseDecoder.decode (bb, cb, true);
                rot13 (cb);
              return (result);
            }
        }
        // --------------------------------------------------------
        /**
         * Unit test for the Rot13 Charset.  This main( ) will open and read
         * an input file if named on the command line, or stdin if no args
         * are provided, and write the contents to stdout via the X-ROT13
         * charset encoding.
         * The "encryption" implemented by the Rot13 algorithm is symmetrical.
         * Feeding in a plain-text file, such as Java source code for example,
         * will output a scrambled version.  Feeding the scrambled version
         * back in will yield the original plain-text document.
         */
        public static void main (String [] argv)
            throws Exception
        {
            BufferedReader in;
            if (argv.length > 0) {
                // Open the named file
                in = new BufferedReader (new FileReader (argv [0]));
            } else {
                // Wrap a BufferedReader around stdin
                in = new BufferedReader (new InputStreamReader (System.in));
            }
            // Create a PrintStream that uses the Rot13 encoding
            PrintStream out = new PrintStream (System.out, false, "X-ROT13");
            String s = null;
            // Read all input and write it to the output.
            // As the data passes through the PrintStream,
            // it will be Rot13-encoded.
            while ((s = in.readLine( )) != null) {
                out.println (s);
            }
            out.flush( );
        }
    }
```

为了使用这个 `Charset` 和它的编码器与解码器，它必须对 Java 运行时环境有效。用 `CharsetProvider` 类完成（示例 6-4）。

示例 6-4.自定义字符集提供方

```
package com.ronsoft.books.nio.charset;

import java.nio.charset.Charset;
import java.nio.charset.spi.CharsetProvider;
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;
/**
 * A CharsetProvider class which makes available the charsets
 * provided by Ronsoft.  Currently there is only one, namely the X-ROT13
 * charset.  This is not a registered IANA charset, so it's
 * name begins with "X-" to avoid name clashes with offical charsets.
 *
 * To activate this CharsetProvider, it's necessary to add a file to
 * the classpath of the JVM runtime at the following location:
 *   META-INF/services/java.nio.charsets.spi.CharsetProvider
 *
 * That file must contain a line with the fully qualified name of
 * this class on a line by itself:
 *   com.ronsoft.books.nio.charset.RonsoftCharsetProvider
```

```
 *
 * See the javadoc page for java.nio.charsets.spi.CharsetProvider
 * for full details.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class RonsoftCharsetProvider extends CharsetProvider
{
    // the name of the charset we provide
    private static final String CHARSET_NAME = "X-ROT13";
   // a handle to the Charset object
    private Charset rot13 = null;
    /**
     * Constructor, instantiate a Charset object and save the reference.
     */
    public RonsoftCharsetProvider(  )
    {
        this.rot13 = new Rot13Charset (CHARSET_NAME, new String [0]);
    }
    /**
     * Called by Charset static methods to find a particular named
     * Charset.  If it's the name of this charset (we don't have
     * any aliases) then return the Rot13 Charset, else return null.
     */
    public Charset charsetForName (String charsetName)
    {
        if (charsetName.equalsIgnoreCase (CHARSET_NAME)) {
            return (rot13);
        }
        return (null);
    }
    /**
     * Return an Iterator over the set of Charset objects we provide.
     * @return An Iterator object containing references to all the
     *  Charset objects provided by this class.
     */
    public Iterator charsets(  )
    {
        HashSet set = new HashSet (1);
        set.add (rot13);
        return (set.iterator(  ));
    }
}
```

对于通过 JVM 运行时环境看到的这个字符集提供方，名为 *META_INF/services/java.nio.charset.spi.CharsetProvider* 的文件必须存在于 JARs 之一内或类路径的目录中。那个文件的内容必须是：

```
com.ronsoft.books.nio.charset.RonsoftCharsetProvider
```

在示例 6-1 中的字符集清单中添加 X-ROT13，产生这个额外的输出：

```
Charset: X-ROT13
  Input:  żMańana?
Encoded:
   0: c2 (Ż)
   1: bf (ż)
   2: 5a (Z)
   3: 6e (n)
   4: c3 (Ă)
   5: b1 (±)
   6: 6e (n)
   7: 61 (a)
   8: 6e (n)
```

```
9: 3f (?)
```

字母 a 和 n 是巧合的分离 13 字母，所以它们在这个特殊的字组中出现在转换位置上。注意非-ASCII 和非字母字符是如何在 UTF-8 编码中保持不变的。

6.4 总结

许多 Java 编程人员永远不会需要处理字符集编码转换问题，而大多数永远不会创建自定义字符集。但是对于那些需要的人，在 *java.nio.charset* 和 *hava.charset.spi* 中的一系列类为字符处理提供了强大的以及弹性的机制。

本章我们学习了 JDK1.4 中新的字符编码的特性。下面是本章的重点：

Charset（字符集类）

封装编码的字符集编码方案，用来表示与作为字节序列的字符集不同的字符序列。

CharsetEncoder（字符集编码类）

编码引擎，把字符序列转化成字节序列。之后字节序列可以被解码从而重新构造源字符序列。

CharsetDecoder（字符集解码器类）

解码引擎，把编码的字节序列转化为字符序列。

CharsetProvider SPI（字符集供应商 SPI）

通过服务器供应商机制定位并使 *Charset* 实现可用，从而在运行时环境中使用。

基本上，我们完成了 NIO 的神奇之旅了。检查一下座位的四周，看看有没有落下的个人物品。非常感谢您的关注，务必要访问我们的网址 http://www.javanio.info/。再见。

附录 A. NIO 与 JNI(Java 本地调用)

The Street finds its own uses for technology.（找寻自我科技出路的街道。）
--Willian Gibson

第二章中曾经提过，直接缓冲器提供了一种方法，通过这种方法 Java 缓冲器对象可以封装系统，或者“原始”内存并使用该内存作为它的后备存储器。在 Java 领域中，您可以通过调用 *ByteBuffer.allocateDirect()*来完成这个操作，它将分配系统内存并在其周围包装一个 Java 对象。

这个方法—分配系统内存并构建封装内存的 Java 对象—是 JDK1.4 中新出现的部分。在之前的版本中，Java 端使用本地码分配的内存是不可能的事情。用 Java 本地调用（JNI）调用的本地码回调到 JVM 并且要求从 JVM 堆中分配内存是可以实现的，但是反方向行不通。Java 代码可以使用以这种方式分配的内存，但是对于本地码如何访问内存有严格的限制。因为上述原因，Java 和本地码共享内存空间难以实现。

在 JDK1.2 中，因为引入了 *GetPrimitiveArrayCrit-ical()*和 *ReleasePrimitiveArrayCritical()*，情况稍有改善。这些新的 JNI 函数让本地码可以更好的控制内存区域。例如，您可以确信在一个重要环节进行的期间，它不会受到垃圾回收器的影响。尽管如此，这些方法也有严格的限制，而分配的内存仍然来自 JVM 堆。

JDK1.4 中的增强带来了三个新的 JNI 函数，反转了这个内存分配模式。JNI 函数 *NewDirectByteBuffer()*拿一个系统内存地址和大小作为参数并构造与返回使用内存区域作为后备存储器的 *ByteBuffer* 对象。这个强大的性能可以实现下面的内容：

- 内存可以被本机码分配，然后包装进缓冲器对象中，通过纯粹的 Java 代码使用。
- 完整的 Java 语义应用到包装缓冲对象上（例如，边界检查、范围划定、垃圾回收等等）。
- 包装对象是 *ByteBuffer*。可以创建它的视窗，可以截取它的片段，可以设置它的字节顺序，并且它可以参与到通道（提供 I/O 可用的潜在的内存空间）上的 I/O 操作中。
- 本地分配的内存不需要包含于 JVM 堆，甚至是 JVM 进程空间内。这样，在指定的内存空间周围包装 *ByteBuffe*r 就可以实现，例如显存或设备控制器。

其他两种新的 JNI 函数简化了 JNI 代码和在 Java 端上创建的直接缓冲器的交互操作。*GetDirectBufferAddress()*和 *GetDirectBufferCapacity()*允许本地码找到直接字节缓冲器的后备存储器的位置与大小。由 *ByteBuffer.allocateDirect()*创建的直接 *ByteBuffer* 对象分配系统内存并且把它包装进对象中（如上所述），但是当 Java 对象被当做垃圾回收时，这些对象也设法释放系统内存。

这就意味着您可以用 *ByteBuffer.allocateDirect()*实例化一个字节缓冲器对象，然后把该对象导入到使用系统内存空间的本地方法中，而不用担心会被垃圾回收扰乱。返回时，Java 代码可以检查缓冲器，得到结果，最后当结束（它将自动的释放相关的系统内存）时允许缓冲器被垃圾回收。这样就减少了复杂性，消除了缓冲器复制，保留了对象的语义（包括垃圾回收），阻止了内存泄露，并且对于您来说需要较少的代码。对于本地码必须进行内存分配的情况，例如获取显存的访问，您需要确定当您完成缓冲器对象时本地码相应的释放内存。

如果您打算与本地码共享直接字节缓冲器，您应该精确的设置到本地字节顺序的缓冲器的字节顺序。基础系统的字节顺序可能和 Java 的字节顺序不一样。即使您不把缓冲器内容视为其他数据类型，这也是一个好主意。它可以让潜在内存的访问更有效。

```
buffer.order (ByteOrder.nativeOrder( ));
```

关于 JNI API 的详细信息，请登陆 Sun 的网站 http://java.sun.con/j2se/1.4/docs/guide/jni/查询。

也许 NIO/JNI 接口的能力最广为流传的，最戏剧化的例子是来自 Jausoft (http://safari.oreilly.com/www.jausoft.com/）的 OpenGL For Java(a.k.a.GL4Java)。该 OpenGL 绑定使用本地 OpenGL 库，不加修改，并且为 OpenGL 提供纯粹的 Java API。它主要是由（脚本产生的）粘着代码组成的，粘着代码把缓冲器引用导入到 OpenGL 库中，只包含极少量的或不包含缓冲器复制。

这样就允许复杂的，实时 3D 应用可以全部用 Java 编写而不用本地码的语句。Sun 创建了使用 OpenGL For Java 的名为 JCanyon 的应用，他们可以在 JavaOne 2001 和 JavaOne 2002 上演示。它是一个实时的，交互的 F16 战斗机模拟器，使用 Grand Canyon 的卫星图勘察地形。它甚至模拟了雾。JCanyon 也充分利用 NIO 通道与存储映射来预取地形数据。整个应用—F16 模拟，地形管理，雾，所有的东西—都是 100%的 Java。OpenGL 通过 GL4Java API 访问—根本不需要系统指定的本地码—而且它在一个标准的笔记本上就能运行。

在 JavaOne2001 到 2002 之间的九个月中，随着 NIO 从进行中到最后的发布，JCanyon 的帧速大约翻了一倍。

JCanyon 的代码是开放的资源，可以从 Sun 的网站 http://java.sun.com/products/jfc/tsc/articles-/jcanyon/上下载。Jausoft OpenGL 绑定，也是开放的资源，跟 JCanyon 代码捆绑在一起，或者您可以直接从 www.jausoft.com/上下载。Jausoft 网站上也有许多小型的程式。

附录 B. 可选择通道 SPI

If you build it, he will come.（如果你建立了它，他就会来。）
--An Iowa Cornfield

可选择通道结构，与 Java 平台的其他几个部分类似，是通过服务器提供者接口（SPI）的方式可插拔的。第六章演示了如何使用可插拔的 *Charset* SPI，而 *Channel* SPI 的运行方式在本质上与 *Charset* SPI 相同。*Channel, Selector,* 甚至是 *SelectionKey* 实现可能是十分复杂的，并且是由操作系统决定的必要条件。示例通道实现超出了本书的范围。附录仅从高层次上概述 SPI。如果您想要开始创建您自己的自定义通道实现，您可能需要本书没有涵盖的更详细的信息。

如我们在第 6.3 节中所看到的，服务是通过低层次的服务机制实例化的提供者类促进的。对于通道来说，基类提供者是 *java.nio.channels.spi.SelectorProvider*。注意这里不叫做 *ChannelProvider*。这个 SPI 仅应用于可选择通道，不是所有的通道类型。在可选择通道与相关选择器（以及选择键，它把一个和另一个连接起来）之间有一种紧密的依赖关系。通道跟来自不同提供者的选择器不能一起运行。

```
package java.nio.channels.spi;
public abstract class SelectorProvider
{
       public static SelectorProvider provider(  )
       // The following methods all throw IOException
       public abstract AbstractSelector openSelector(  )
       public abstract ServerSocketChannel openServerSocketChannel(  )
       public abstract SocketChannel openSocketChannel(  )
       public abstract DatagramChannel openDatagramChannel(  )
       public abstract Pipe openPipe(  )
}
```

SelectorProvider 实例公开工厂方法创建具体 *Selector* 对象、套接通道对象的三种类型，以及 *Pipe* 对象。产生的 *Selector* 对象必须和来自同一个提供者（和来自该提供者的 *Pipe* 对象产生的通道）的套接通道对象交互运行。来自给定提供者的通道，选择器，以及选择键有权使用彼此的内部实现细节。

SelectorProvider 的 API 相当的明显，带有上面列出的首个方法的可能异常。*provider()*方法是 *SelectorProvider* 上的类方法，返回一个引用到默认系统提供者上。在 JVM 启动时确定默认提供者，方式跟 *CharsetProvider* 对象相同（详情请参阅 *SelectorProvider* 的 Java 开发文档）。您可以通过您选择的其他方式获得 *SelectorProvider*（例如，在被允许的情况下，直接对它实例化），然后直接在提供者对象上调用实例工厂方法。通道类的默认工厂方法，例如 *SocketChannel.open()*，在默认 *SelectorProvider* 对象上直接穿过相应的工厂方法。

`java.nio.channels.spi` 包中有四种其他的类：*AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelectionKey,* 和 *Abstract-Selector*。您可能记得图 3-2 中，*AbstractSelectableChannel* 经由 *java.nio.channels.Selectable-Channel* 延展 *AbstractInterruptibleChannel*。*AbstractInterruptibleChannel* 提供一个共同的框架，用来管理可能被中断的通道，而 *AbstractSelectableChannel* 为可选择通道提供了类似的支持。因为 *AbstractInterruptibleChannel* 是 *AbstractSelectableChannel* 的原形，所有的可选择通道都是可中断

的。

```
package java.nio.channels.spi;
public abstract class AbstractInterruptibleChannel
       implements Channel, InterruptibleChannel
{
       protected AbstractInterruptibleChannel(  )
       public final void close(  ) throws IOException
       public final boolean isOpen(  )
       protected final void begin(  )
       protected final void end (boolean completed)
       protected abstract void implCloseChannel(  ) throws IOException;
}
public abstract class AbstractSelectableChannel
       extends java.nio.channels.SelectableChannel
{
       protected AbstractSelectableChannel (SelectorProvider provider)
       public final SelectorProvider provider(  )
       public final boolean isRegistered(  )
       public final SelectionKey keyFor (Selector sel)
       public final SelectionKey register (Selector sel,
              int ops, Object att)
       public final boolean isBlocking(  )
       public final Object blockingLock(  )
       public final SelectableChannel configureBlocking (boolean block)
              throws IOException
       protected final void implCloseChannel(  )
              throws IOException
       protected abstract void implCloseSelectableChannel(  )
              throws IOException;
       protected abstract void implConfigureBlocking (boolean block)
              throws IOException;
}
```

任何需要支持选择的通道实现都必须扩展 *AbstractSelectableChannel*。上面列出的两个类提供的大部分方法都是您已经见过的默认实现，都在第三章通道类的 API 中见过。每一个都有少量的受保护方法，可以（或必须）实现子类用来创建新的可选择通道类的方法。

其他的两个类，*AbstractSelector* 和 *AbstractSelectionKey*，提供来自具体实现类必须扩展的类似的模块：

```
package java.nio.channels.spi;
public abstract class AbstractSelector
       extends Selector
{
       protected AbstractSelector(SelectorProvider provider)
       public final void close(  ) throws IOException
       public final boolean isOpen(  )
       public final SelectorProvider provider(  )
       protected abstract SelectionKey register (
              AbstractSelectableChannel ch, int ops, Object att);
       protected final void deregister (AbstractSelectionKey key)
       protected final Set cancelledKeys(  )
       protected final void begin(  )
       protected final void end(  )
       protected abstract void implCloseSelector(  ) throws IOException;
}
public abstract class AbstractSelectionKey
       extends SelectionKey
{
       protected AbstractSelectionKey(  )
       public final boolean isValid(  )
       public final void cancel(  )
```

```
}
```

再次的，您能看到公共方法和只有子类使用的受保护方法的默认执行。在 *AbstractSelector* 中，您可以发现取消键和外在的通道注销注册的内在联系。

只有极少数的“民间”Java 编程人员需要关心通道 SPI。它是主要被 JVM 卖主和/或高端产品的卖主，例如应用服务器，占据的领域。创建新的可选择通道实现是一项非凡的事业，它需要专业技能和大量的资源。

附录 C. NIO 快速参考

I still haven't found what I'm looking for.（我还没有找到我所寻找的。）
--U2

该附录是 NIO 类和接口、包、类，以及方法的快速参考指南，按字母顺序分类从而易于查找所需信息。API 清单是使用 Java 反射 API 从 JDK 的编译类文件中直接抽取信息而编程创建的。正规表达式（类信息中自动产生的）被用来找回资源中的参数名称。说明性文字（例如这个）在 XML 中构成，之后由 XSL 样式表进行处理，根据每个类合并 API 信息（通过调用 Java 代码作为扩展函数）。

此处使用的惯例与正文相同。在方法签名的结束部分缺少分号表示资源的后部有方法体。抽取方法以分号结尾因为他们没有具体的主体。未列出到已初始化的常量的值。

本参考不适用于 J2SE 1.4.0 版本。

C.1 java.nio 包

`java.nio` 包包含在 `java.nio.channels` 和 `java.nio.charset` 分包中的类使用的 `Buffer` 类。

C.1.1 Buffer

`Buffer` 是所有其他缓冲类延展出来的基类。它包含所有缓冲器类型都适用的通用方法。

```
public abstract class Buffer
{
    public final int   capacity()
    public final Buffer clear()
    public final Buffer flip()
    public final boolean hasRemaining()
    public abstract boolean isReadOnly();
    public final int limit()
    public final Buffer limit (int newLimit)
    public final Buffer mark()
    public final int position()
    public final Buffer position (int newPosition)
    public final int remaining()
    public final Buffer reset()
    public final Buffer rewind()
}
```

参见：第 C.1.4 节

C.1.2 BufferOverflowException

当一个简单相关 *put()*试图用缓冲器的位置相等于它的极限时，或当批量 *put()*引起位置超出限度时抛出 `BufferOverflowException`（未检查）。

```
public class  BufferOverflowException
   extends RuntimeException
{
   public  BufferOverflowException()
}
```

参见：第 C.1.1 节

C.1.3 BufferUnderflowException

当一个简单相关 *get()*试图用缓冲器的位置相等于它的极限时，或当批量 *get()*引起位置超出限度时抛出 `BufferUnderflowException`（未检查）。

```
public class  BufferUnderflowException
   extends RuntimeException
{
   Public BufferUnderflowException()
}
```

参见：第 C.1.1 节

C.1.4 ByteBuffer

`ByteBuffer` 是所有缓冲器类中最复杂也最万能的类。只有字节缓冲器可以在通道上参与到 I/O 操作中，发送与接收数据。

```
public abstract class ByteBuffer
   extends Buffer
   implements Comparable
{
   public static ByteBuffer  allocate (int capacity)
   public static ByteBuffer  allocateDirect (int capacity)
   public final byte []array()
   public final int arrayOffset()
   public abstract CharBuffer  asCharBuffer();
   public abstract DoubleBuffer  asDoubleBuffer();
   public abstract FloatBuffer  asFloatBuffer();
   public abstract IntBuffer asIntBuffer();
   public abstract LongBuffer  asLongBuffer();
   public abstract ByteBuffer  asReadOnlyBuffer();
   public abstract ShortBuffer  asShortBuffer();
   public abstract ByteBuffer  compact();
   public int compareTo (Object ob)
   public abstract ByteBuffer  duplicate();
   public boolean equals (Object ob)
   public abstract byte get();
   public ByteBuffer get (byte [] dst)
   public abstract byte     get (int index);
   public ByteBuffer get (byte [] dst, int offset, int length)
   public abstract char     getChar();
   public abstract char getChar (int index);
   public abstract double  getDouble();
   public abstract double  getDouble (int index);
   public abstract float getFloat();
   public abstract float  getFloat (int index);
   public abstract int getInt();
   public abstract int getInt (int index);
   public abstract long getLong();
   public abstract long getLong (int index);
```

```
    public abstract short getShort();
    public abstract short getShort (int index);
    public final boolean hasArray()
    public int hashCode()
    public abstract boolean isDirect();
    public final ByteOrder order()
    public final ByteBuffer order (ByteOrder bo)
    public abstract ByteBuffer put (byte b);
    public final ByteBuffer put (byte [] src)
    public ByteBuffer put (ByteBuffer src)
    public abstract ByteBuffer put (int index, byte b);
    public ByteBuffer put (byte [] src, int offset, int length)
    public abstract ByteBuffer putChar (char value);
    public abstract ByteBuffer putChar (int index, char value);
    public abstract ByteBuffer putDouble (double value);
    public abstract ByteBuffer putDouble (int index, double value);
    public abstract ByteBuffer putFloat (float value);
    public abstract ByteBuffer putFloat (int index, float value);
    public abstract ByteBuffer putInt (int value);
    public abstract ByteBuffer putInt (int index, int value);
    public abstract ByteBuffer putLong (long value);
    public abstract ByteBuffer putLong (int index, long value);
    public abstract ByteBuffer putShort (short value);
    public abstract ByteBuffer putShort (int index, short value);
    public abstract ByteBuffer slice();
    public String toString()
    public static ByteBuffer wrap (byte [] array)
    public static ByteBuffer wrap (byte [] array, int offset, int length)
}
```

参见：第 C.1.1 节，第 C.1.5 节

C.1.5 **ByteOrder**

ByteOrder 是安全类型的列举，它不能被直接实例化。ByteOrder 的两个公开可访问实例被视为静态类域。提供一个类方法用来确定基础操作系统的本地字节顺序，可以跟 Java 平台默认的不同。

```
public final class ByteOrder
{
    public static final ByteOrder  BIG_ENDIAN
    public static final ByteOrder LITTLE_ENDIAN
    public static ByteOrder nativeOrder()
    public String toString()
}
```

参见：第C.1.4节

C.1.6 **CharBuffer**

CharBuffer 管理类型 char 的数据元素并实现 CharSequence 接口，允许其参与到字符为导向的操作中，例如正规表达式匹配。在 java.nio.charset 包中类也使用 CharBuffer。

```
public abstract class CharBuffer
    extends Buffer
    implements Comparable, CharSequence
{
    public static CharBuffer allocate (int capacity)
    public final char [] array()
    public final int arrayOffset()
```

```
    public abstract CharBuffer asReadOnlyBuffer();
    public final char charAt (int index)
    public abstract CharBuffer compact();
    public int compareTo (Object ob)
    public abstract CharBuffer duplicate();
    public boolean  equals (Object ob)
    public abstract char get();
    public CharBuffer get(char [] dst)
    public abstract char get (int index);
    public CharBuffer get (char [] dst, int offset, int length)
    public final boolean hasArray()
    public int hashCode()
    public abstract boolean isDirect();
    public final int length()
    public abstract ByteOrder order();
    public abstract CharBuffer put (char c);
    public final CharBuffer put (char [] src)
    public final CharBuffer put (String src)
    public CharBuffer put (CharBuffer src)
    public abstract CharBuffer put (int index, char c);
    public CharBuffer put (char [] src, int offset, int length)
    public CharBuffer put (String src, int start, int end)
    public abstract CharBuffer slice();
    public abstract CharSequence subSequence (int start, int end);
    public String toString()
    public static CharBuffer wrap (char [] array)
    public static CharBuffer wrap (CharSequence csq)
    public static CharBuffer wrap (char [] array, int offset, int length)
    public static CharBuffer wrap (CharSequence csq, int start, int end)
}
```

参见：第 C.1.1 节，`java.lang.CharSequence, java.util.regex.Matcher`

C.1.7 **DoubleBuffer**

```
public abstract class DoubleBuffer
    extends Buffer
    implements Comparable
{
    public static DoubleBuffer allocate (int capacity)
    public final double [] array()
    public final int arrayOffset()
    public abstract DoubleBuffer asReadOnlyBuffer();
    public abstract DoubleBuffer compact();
    public int compareTo (Object ob)
    public abstract DoubleBuffer duplicate();
    public boolean  equals (Object ob)
    public abstract double get();
    public DoubleBuffer get (double [] dst)
    public abstract double get (int index);
    public DoubleBuffer get (double [] dst, int offset, int length)
    public final boolean hasArray()
    public int  hashCode()
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public abstract DoubleBuffer put (double d);
    public final DoubleBuffer put (double [] src)
    public DoubleBuffer put (DoubleBuffer src)
    public abstract DoubleBuffer put (int index, double d);
    public DoubleBuffer put (double [] src, int offset, int length)
    public abstract DoubleBuffer slice();
    public String toString()
    public static DoubleBuffer wrap (double [] array)
    public static DoubleBuffer wrap (double [] array, int offset,
             int length)
```

```
}
```

参见：第 C.1.1 节

C.1.8 FloatBuffer

```
FloatBuffer manages data elements of type float.

public abstract class FloatBuffer
   extends Buffer
   implements Comparable
{
   public static FloatBuffer allocate (int capacity)
   public final float [] array()
   public final int arrayOffset()
   public abstract FloatBuffer  asReadOnlyBuffer();
   public abstract FloatBuffer  compact();
   public int  compareTo (Object ob)
   public abstract FloatBuffer   duplicate();
   public boolean  equals (Object ob)
   public abstract float get();
   public FloatBuffer get (float [] dst)
   public abstract float get (int index);
   public FloatBuffer get (float [] dst, int offset, int length)
   public final boolean hasArray()
   public int  hashCode()
   public abstract boolean isDirect();
   public abstract ByteOrder order();
   public abstract FloatBuffer  put (float f);
   public final FloatBuffer  put (float [] src)
   public FloatBuffer put (FloatBuffer src)
   public abstract FloatBuffer  put (int index, float f);
   public FloatBuffer put (float [] src, int offset, int length)
   public abstract FloatBuffer  slice();
   public String toString()
   public static FloatBuffer wrap (float [] array)
   public static FloatBuffer wrap (float [] array, int offset, int
length)
}
```

参见：第 C.1.1 节

C.1.9 IntBuffer

```
IntBuffer manages data elements of type int.

public abstract class IntBuffer
   extends Buffer
   implements Comparable
{
   public static IntBuffer allocate (int capacity)
   public final int [] array()
   public final int arrayOffset()
   public abstract IntBuffer asReadOnlyBuffer();
   public abstract IntBuffer compact();
   public int compareTo (Object ob)
   public abstract IntBuffer  duplicate();
   public boolean  equals (Object ob)
   public abstract int get();
   public abstract int get (int index);
   public IntBuffer get (int [] dst)
   public IntBuffer get (int [] dst, int offset, int length)
```

```
    public final boolean hasArray()
    public int  hashCode()
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public abstract IntBuffer put (int i);
    public final IntBuffer put (int [] src)
    public IntBuffer put (IntBuffer src)
    public abstract IntBuffer put (int index, int i);
    public IntBuffer put (int [] src, int offset, int length)
    public abstract IntBuffer slice();
    public String toString()
    public static IntBuffer wrap (int [] array)
    public static IntBuffer wrap (int [] array, int offset, int length)
}
```

参见：第 C.1.1 节

C.1.10 **InvalidMarkException**

```
InvalidMarkException (unchecked) is thrown when reset() is invoked on a
buffer that does
not have a mark set.

public class InvalidMarkException
    extends IllegalStateException
{
    Public InvalidMarkException()
}
```

参见：第 C.1.1 节

C.1.11 **LongBuffer**

```
LongBuffer manages data elements of type long.
public abstract class LongBuffer
    extends Buffer
    implements Comparable
{
    public static LongBuffer allocate (int capacity)
    public final long [] array()
    public final int arrayOffset()
    public abstract LongBuffer asReadOnlyBuffer();
    public abstract LongBuffer compact();
    public int compareTo (Object ob)
    public abstract LongBuffer duplicate();
    public boolean  equals (Object ob)
    public abstract long get();
    public abstract long get (int index);
    public LongBuffer get (long [] dst)
    public LongBuffer get (long [] dst, int offset, int length)
    public final boolean hasArray()
    public int  hashCode()
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public LongBuffer put (LongBuffer src)
    public abstract LongBuffer put (long l);
    public final LongBuffer put (long [] src)
    public abstract LongBuffer put (int index, long l);
    public LongBuffer put (long [] src, int offset, int length)
    public abstract LongBuffer slice();
    public String toString()
    public static LongBuffer wrap (long [] array)
```

```
    public static LongBuffer wrap (long [] array, int offset, int length)
}
```

参见：第 C.1.1 节

C.1.12 **MappedByteBuffer**

MappedByteBuffer 是 ByteBuffer 的特殊类型，它的数据元素是硬盘文件的内容。MappedByteBuffer 对象只能通过调用 FileChannel 对象的 *map()*方法创建。

```
public abstract class MappedByteBuffer
    extends ByteBuffer
{
    public final MappedByteBuffer force()
    public final boolean isLoaded()
    public final MappedByteBuffer load()
}
```

参见：第 C.1.1 节，第 C.2.12 节

C.1.13 **ReadOnlyBufferException**

当有方法将更改缓冲器内容时，例如 *put()*或 *compact()*，抛出 ReadOnlyBufferException（未检查），在只读缓冲器上调用。

```
public class ReadOnlyBufferException
    extends UnsupportedOperationException
{
    public ReadOnlyBufferException()
}
```

参见：第 C.1.1 节

C.1.14 **ShortBuffer**

ShortBuffer 管理类型 short 的数据元素。

```
ShortBuffer manages data elements of type short.

public abstract class ShortBuffer
    extends Buffer
    implements Comparable
{
    public static ShortBuffer allocate (int capacity)
    public final short [] array()
    public final int arrayOffset()
    public abstract ShortBuffer asReadOnlyBuffer();
    public abstract ShortBuffer compact();
    public int compareTo (Object ob)
    public abstract ShortBuffer duplicate();
    public boolean equals (Object ob)
    public abstract short get();
    public abstract short get (int index);
    public ShortBuffer get (short [] dst)
    public ShortBuffer get (short [] dst, int offset, int length)
```

```
    public final boolean hasArray()
    public int  hashCode()
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public ShortBuffer  put (ShortBuffer src)
    public abstract ShortBuffer put (short s);
    public final ShortBuffer put (short [] src)
    public abstract ShortBuffer put (int index, short s);
    public ShortBuffer put (short [] src, int offset, int length)
    public abstract ShortBuffer slice();
    public String toString()
    public static ShortBuffer wrap (short [] array)
    public static ShortBuffer wrap (short [] array, int offset, int
length)
}
```

参见：第 C.1.1 节

C.2 **java.nio.channels 包**

java.nio.channels 包包含与通道和选择器相关的类和接口。

C.2.1 **AlreadyConnectedException**

当在已连接的 SocketChannel 对象上调用 *connect()*时抛出 AlreadyConnectedException（未检查）。

```
public class  AlreadyConnectedException
    extends IllegalStateException
{
    public AlreadyConnectedException()
}
```

参见：java.net.socket, 第 C.2.32 节

C.2.2 **AsynchronousCloseException**

当线程被锁定在通道操作上时，例如 *read()*和 *write()*，抛出 AsynchronousCloseException（IOException 的子类），并且通道被另一个线程关闭。

```
public class  AsynchronousCloseException
    extends ClosedChannelException
{
    public AsynchronousCloseException()
}
```

参见：第 C.2.7 节

C.2.3 **ByteChannel**

ByteChannel 是一个空聚合接口。它把 ReadableByteChannel 和 WritableByteChannel 结合到一个单接口中，但是不定义任何新的方法。

```
public interface ByteChannel
   extends ReadableByteChannel, WritableByteChannel
{
}
```

参见：第 C.2.5 节，第 C.2.26 节，第 C.2.35 节

C.2.4 **CancelledKeyException**

当试图使用已失效的 SelectionKey 对象时抛出 CancelledKeyException（未检查）。

```
public class CancelledKeyException
   extends IllegalStateException
{
   public  CancelledKeyException()
}
```

参见：第 C.2.29 节，第 C.2.30 节

C.2.5 **Channel**

Channel 是所有其他通道接口的超接口。它定义所有具体通道类的共同方法。

```
public interface Channel
{
   public void  close()
      throws java.io.IOException;
   public boolean  isOpen();
}
```

参见：第 C.2.3 节，第 C.2.15 节，第 C.2.26 节，第 C.2.27 节，第 C.2.35 节

C.2.6 **Channels**

Channels 是一个效用类，使通道可以交互运行传统的字节与字符流。工厂方法返回包装对象，对应通道到流上，反之亦然。返回的通道对象可能不是可选的，也不是中断的。

```
public final class Channels
{
   public static ReadableByteChannel newChannel (java.io.InputStream in)
   public static WritableByteChannel newChannel (java.io.OutputStream 
out)
   public static java.io.InputStream newInputStream (
      ReadableByteChannel ch)
   public static java.io.OutputStream newOutputStream (
      WritableByteChannel ch)
   public static java.io.Reader  newReader (ReadableByteChannel ch,
      String csName)
   public static java.io.Reader  newReader (ReadableByteChannel ch,
      java.nio.charset.CharsetDecoder dec, int minBufferCap)
   public static java.io.Writer  newWriter (WritableByteChannel ch,
      String csName)
   public static java.io.Writer  newWriter (WritableByteChannel ch,
```

```
        java.nio.charset.CharsetEncoder enc, int minBufferCap)
}
```

参见：第 C.4.3 节，第 C.4.4 节，java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer，第 C.2.26 节，第 C.2.35 节

C.2.7 **ClosedByInterruptException**

当线程在通道操作上被阻塞，例如 *read()*或 *write()*，以及被另一个线程中断时抛出 ClosedByInterruptException（IOException 的子类）。休眠线程上的通道将作为副作用被关闭。这个异常与 AsynchronousCloseException 类似，但它是当休眠线程直接被中断时的结果。

```
public class ClosedByInterruptException
   extends AsynchronousCloseException
{
   public  ClosedByInterruptException()
}
```

参见：第 C.2.2 节, java.lang.Thread

C.2.8 **ClosedChannelException**

当试图在已经关闭的通道上操作时抛出 ClosedChannelException（IOException 的子类）。一些通道，例如 SocketChannel，可能针对一些操作关闭但不是全部的操作。例如，SocketChannel 的每一面都可能自主关闭，而其他的继续正常运行。

```
public class ClosedChannelException
   extends java.io.IOException
{
   public  ClosedChannelException()
}
```

参见：第 C.2.5 节

C.2.9 **ClosedSelectorException**

当试图使用已经关闭的 Selector 时抛出 ClosedSelectorException（未检查）。

```
public class ClosedSelectorException
   extends IllegalStateException
{
   public  ClosedSelectorException()
}
```

参见：第 C.2.30 节

C.2.10 **ConnectionPendingException**

当在非阻塞模式中，因为并发连接正在进行中，在 SocketChannel 对象上调用 *connect()*时抛出 ConnectionPendingException（未检查）。

```
public class  ConnectionPendingException
   extends IllegalStateException
{
   public  ConnectionPendingException()
}
```

参见：第 C.2.32 节

C.2.11 **DatagramChannel**

DatagramChannel 类分别提供来自和到 ByteBuffer 对象的数据包的发送与接收。

```
public abstract class DatagramChannel
   extends java.nio.channels.spi.AbstractSelectableChannel
   implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
   public abstract DatagramChannel connect (java.net.SocketAddress
remote)
       throws java.io.IOException;
   public abstract DatagramChannel disconnect()
      throws java.io.IOException;
   public abstract boolean isConnected();
   public static DatagramChannel open()
      throws java.io.IOException
   public abstract int read (java.nio.ByteBuffer dst)
      throws java.io.IOException;
   public final long read (java.nio.ByteBuffer [] dsts)
      throws java.io.IOException
   public abstract long read (java.nio.ByteBuffer [] dsts, int offset,
     int length) throws java.io.IOException;
   public abstract java.net.SocketAddress receive (
     java.nio.ByteBuffer dst) throws java.io.IOException;
   public abstract int send (java.nio.ByteBuffer src,
     java.net.SocketAddress target) throws java.io.IOException;
   public abstract java.net.DatagramSocket socket();
   public final int validOps()
   public abstract int write (java.nio.ByteBuffer src)
      throws java.io.IOException;
   public final long write (java.nio.ByteBuffer [] srcs)
      throws java.io.IOException
   public abstract long write (java.nio.ByteBuffer [] srcs, int offset,
     int length) throws java.io.IOException;
}
```

参见：java.net.DatagramSocket

C.2.12 **FileChannel**

FileChannel 类提供了丰富的文件为导向的操作。只有在 RandomAccessFile, FileInputStream，或者 FileOutputStream 对象上调用 *getChannel()*方法才能获得 FileChannel 对象。

```
public abstract class FileChannel
    extends java.nio.channels.spi.AbstractInterruptibleChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
    public abstract void force (boolean metaData)
        throws java.io.IOException;
    public final FileLock lock()
        throws java.io.IOException
    public abstract FileLock lock (long position, long size,
       boolean shared) throws java.io.IOException;
    public abstract java.nio.MappedByteBuffer map (
       FileChannel.MapMode mode, long position, long size)
        throws java.io.IOException;
    public abstract long position()
        throws java.io.IOException;
    public abstract FileChannel position (long newPosition)
        throws java.io.IOException;
    public abstract int read (java.nio.ByteBuffer dst)
        throws java.io.IOException;
    public final long read (java.nio.ByteBuffer [] dsts)
        throws java.io.IOException
    public abstract int read (java.nio.ByteBuffer dst, long position)
        throws java.io.IOException;
    public abstract long read (java.nio.ByteBuffer [] dsts, int offset,
       int length) throws java.io.IOException;
    public abstract long size()
        throws java.io.IOException;
    public abstract long transferFrom (ReadableByteChannel src,
       long position, long count) throws java.io.IOException;
    public abstract long transferTo (long position, long count,
        WritableByteChannel target) throws java.io.IOException;
    public abstract FileChannel truncate (long size)
        throws java.io.IOException;
    public final FileLock tryLock()
        throws java.io.IOException
    public abstract FileLock tryLock (long position, long size,
       boolean shared) throws java.io.IOException;
    public abstract int write (java.nio.ByteBuffer src)
        throws java.io.IOException;
    public final long write (java.nio.ByteBuffer [] srcs)
        throws java.io.IOException
    public abstract int write (java.nio.ByteBuffer src, long position)
        throws java.io.IOException;
    public abstract long write (java.nio.ByteBuffer [] srcs, int offset,
       int length) throws java.io.IOException;
    public static class FileChannel.MapMode
    {
        public static final FileChannel.MapMode PRIVATE
        public static final FileChannel.MapMode READ_ONLY
        public static final FileChannel.MapMode READ_WRITE
        public String toString()
    }
}
```

参见：java.io.FileInputStream, java.io.FileOutputStream, java.io.RandomAccessFile，第 C.1.12 节

C.2.13 **FileLock**

FileLock 类封装与 FileChannel 对象相关的模块区。

```
public abstract class FileLock
```

```
{
   public final FileChannel channel()
   public final boolean isShared()
   public abstract boolean isValid();
   public final boolean overlaps (long position, long size)
   public final long position()
   public abstract void release()
      throws java.io.IOException;
   public final long size()
   public final String toString()
}
```

参见：第 C.2.12 节

C.2.14 **FileLockInterruptionException**

当阻塞的线程等待授权的文件锁被另一个线程中断时抛出 `FIleLockInterruptionException`。`FileChannel` 并没有关闭，但是经由捕捉这个异常，设置中断的线程的中断状态。如果未清除它的中断状态（通过调用 *Thread.interrupted()*），它将引起下一个触碰的通道的关闭。

```
public interface       GatheringByteChannel
   extends WritableByteChannel
{
   public long write (java.nio.ByteBuffer [] srcs)
      throws java.io.IOException;
   public longwrite (java.nio.ByteBuffer [] srcs, int offset, int length)
      throws java.io.IOException;
}
```

参见：第 C.2.3 节，第 C.2.26 节，第 C.2.27 节，第 C.2.35 节

C.2.15 **GatheringByteChannel**

GatheringByteChannel 接口定义了汇集数据写入到一个通道中的方法。

```
public interface GatheringByteChannel
    extends WritableByteChannel
{
    public long write (java.nio.ByteBuffer [] srcs)
    throws java.io.IOException;
    public long write (java.nio.ByteBuffer [] srcs, int offset, int length)
    throws java.io.IOException;
}
```

参见：第 C.2.3 节，第 C.2.26 节，第 C.2.27 节，第 C.2.35 节

C.2.17 **IllegalSelectorException**

当试图用来自不同 `SelectorProvider` 类的 `Selector` 注册 `SelectableChannel` 时抛出 `IllegalSelectorException`（未检查）。选择器只在使用相同提供者创建的通道时才能正常运行。

```
public class IllegalSelectorException
   extends IllegalArgumentException
{
   public  IllegalSelectorException()
}
```

参见：第 C.3.5 节

C.2.18 **InterruptibleChannel**

InterruptibleChannel 是标记接口，如果实现，表示通道类是中断的。所有的可选择通道都是中断的。

```
public interface InterruptibleChannel
   extends Channel
{
   public void  close()
      throws java.io.IOException;
}
```

参见：第 C.2.28 节

C.2.19 **NoConnectionPendingException**

在非阻塞模式中，也就是之前未调用 *connect()*开始并发的连接过程，在 SocketChannel 对象上调用 *finishiConnect()*时抛出 NoConnectionPendingException（未检查）。

```
public class NoConnectionPendingException
   extends IllegalStateException
{
   public  NoConnectionPendingException()
}
```

参见：第 C.2.32 节

C.2.20 **NonReadableChannelException**

当在读取允许没有开启的通道上调用 *read()*方法时抛出 NonReadableChannelException（未检查）。

```
public class NonReadableChannelException
   extends IllegalStateException
{
   public  NonReadableChannelException()
}
```

参见：第 C.2.26 节

C.2.21 **NonWritableChannelException**

当在写允许没有开启的通道上调用 *write()*方法时抛出 NonWritableChannelException（未检查）。

```
public class NonWritableChannelException
    extends IllegalStateException
{
    public NonWritableChannelException()
}
```

参见：第 C.2.35 节

C.2.22 **NotYetBoundException**

当试图执行一个操作时，例如 *accept()*，在尚未被绑定到端口的 ServerSocketChannel 上，抛出 NotYetBoundException（未检查）。

```
public class NotYetBoundException
    extends IllegalStateException
{
    public NotYetBoundException()
}
```

参见：java.net.ServerSocket

C.2.23 **NotYetConnectedException**

当在调用 *connect()*之前或并发的连接成功完成之前，试图使用 I/O 的 SocketChannel 对象时抛出 NotYetConnectedException（未检查）。

```
public class NotYetConnectedException
    extends IllegalStateException
{
    public NotYetConnectedException()
}
```

参见：第 C.2.32 节

C.2.24 **OverlappingFileLockException**

当试图获取已经被同一个 JVM 上锁的文件区域上的锁时，或者当另一个线程正在等待给属于同一个文件的覆盖区域上锁时抛出 OverlappingFileLockException（未检查）。

```
public class OverlappingFileLockException
    extends IllegalStateException
{
    public OverlappingFileLockException()
}
```

参见：第 C.2.12 节，第 C.2.13 节

C.2.25 **Pipe**

Pipe 是包含一对可选择通道的聚合类。这些通道交叉相连构成一个回送。SinkChannel 对象是管道的写结尾；无论写什么，都是在 SourceChannel 对象上读取有效。

```
public abstract class Pipe
{
   public static Pipe open()
      throws java.io.IOException
   public abstract Pipe.SinkChannel  sink();
   public abstract Pipe.SourceChannel  source();
   public abstract static class  Pipe.SinkChannel
      extends java.nio.channels.spi.AbstractSelectableChannel
      implements WritableByteChannel, GatheringByteChannel
   {
      public final int validOps()
   }
   public abstract static class  Pipe.SourceChannel
      extends java.nio.channels.spi.AbstractSelectableChannel
      implements ReadableByteChannel, ScatteringByteChannel
   {
      public final int validOps()
   }
}
```

参见：第 C.2.28 节，第 C.2.30 节

C.2.26 **ReadableByteChannel**

ReadableByteChannel 接口定义 *read()*方法，实现了通道从 ByteBuffer 对象内的通道上读取数据的操作。

```
public interface ReadableByteChannel
   extends Channel
{
   public int  read (java.nio.ByteBuffer dst)
      throws java.io.IOException;
}
```

参见：第 C.1.4 节，第 C.2.3 节，第 C.2.35 节

C.2.27 **ScatteringByteChannel**

ScatteringByteChannel 接口定义执行来自通道的散射读取的方法。

```
public interface  ScatteringByteChannel
   extends ReadableByteChannel
{
   public long  read (java.nio.ByteBuffer [] dsts)
      throws java.io.IOException;
   public long read (java.nio.ByteBuffer [] dsts, int offset, int length)
```

```
        throws java.io.IOException;
}
```

参见：第 C.2.3 节，第 C.2.15 节，第 C.2.26 节，第 C.2.35 节，

C.2.28 **SelectableChannel**

SelectableChannel 是所有能够参加 Selector 对象控制的选择操作的通用超类。SelectableChannel 对象可以被放置非阻塞模式中，并且在非阻塞模式期间只能用 Selector 进行注册。所有从 SelectableChannel 扩展的类也都实现 InterruptibleChannel。

```
public abstract class SelectableChannel
    extends java.nio.channels.spi.AbstractInterruptibleChannel
    implements Channel
{
    public abstract Object blockingLock();
    public abstract SelectableChannel configureBlocking (boolean block)
        throws java.io.IOException;
    public abstract boolean isBlocking();
    public abstract boolean isRegistered();
    public abstract SelectionKey keyFor (Selector sel);
    public abstract java.nio.channels.spi.SelectorProvider provider();
    public final SelectionKey register (Selector sel, int ops)
        throws ClosedChannelException
    public abstract SelectionKey register (Selector sel, int ops,
       Object att) throws ClosedChannelException;
    public abstract int validOps();
}
```

参见：第 C.2.30 节

C.2.29 **SelectionKey**

SelectionKey 通过 Selector 对象封装 SelectableChannel 对象的注册。

```
public abstract class SelectionKey
{
    public static final int OP_ACCEPT
    public static final int OP_CONNECT
    public static final int OP_READ
    public static final int OP_WRITE
    public final Object attach (Object ob)
    public final Object attachment()
    public abstract void cancel();
    public abstract SelectableChannel channel();
    public abstract int interestOps();
    public abstract SelectionKey interestOps (int ops);
    public final boolean isAcceptable()
    public final boolean isConnectable()
    public final boolean isReadable()
    public abstract boolean isValid();
    public final boolean isWritable()
    public abstract int readyOps();
    public abstract Selector selector();
}
```

参见：第 C.2.28 节，第 C.2.30 节

C.2.30 **Selector**

Selector 是协调类，执行已注册的 SelectableChannel 对象的就绪选择，并管理相关的键和状态信息。

```
public abstract class Selector
{
   public abstract void close()
      throws java.io.IOException;
   public abstract boolean isOpen();
   public abstract java.util.Set  keys();
   public static Selector open()
      throws java.io.IOException
   public abstract java.nio.channels.spi.SelectorProvider provider();
   public abstract int select()
      throws java.io.IOException;
   public abstract int select (long timeout)
      throws java.io.IOException;
   public abstract int selectNow()
      throws java.io.IOException;
   public abstract java.util.Set  selectedKeys();
   public abstract Selector  wakeup();
}
```

参见：第 C.2.28 节，第 C.2.29 节

C.2.31 **ServerSocketChannel**

ServerSocketChannel 类侦听引入的套接连接并且创建新的 SocketChannel 实例。

```
public abstract class ServerSocketChannel
   extends java.nio.channels.spi.AbstractSelectableChannel
{
   public abstract SocketChannel  accept()
      throws java.io.IOException;
   public static ServerSocketChannel open()
      throws java.io.IOException
   public abstract java.net.ServerSocket socket();
   public final int validOps()
}
```

参见：java.net.InetSocketAddress, java.net.ServerSocket. java.net.SocketAddress, 第 C.2.32 节

C.2.32 **SocketChannel**

SocketChannel 对象在字节缓冲器和网络连接之间转化数据。

```
public abstract class SocketChannel
   extends java.nio.channels.spi.AbstractSelectableChannel
   implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
{
   public abstract boolean connect (java.net.SocketAddress remote)
      throws java.io.IOException;
   public abstract boolean finishConnect()
      throws java.io.IOException;
```

```
        public abstract boolean isConnected();
        public abstract boolean isConnectionPending();
        public static SocketChannel open()
           throws java.io.IOException
        public static SocketChannel open (java.net.SocketAddress remote)
           throws java.io.IOException
        public abstract int read (java.nio.ByteBuffer dst)
           throws java.io.IOException;
        public final long read (java.nio.ByteBuffer [] dsts)
           throws java.io.IOException
        public abstract long read (java.nio.ByteBuffer [] dsts, int offset,
          int length) throws java.io.IOException;
        public abstract java.net.Socket  socket();
        public final int validOps()
        public abstract int write (java.nio.ByteBuffer src)
           throws java.io.IOException;
        public final long write (java.nio.ByteBuffer [] srcs)
           throws java.io.IOException
        public abstract long write (java.nio.ByteBuffer [] srcs, int offset,
          int length) throws java.io.IOException;
    }
```

参见：java.net.InetSocketAddress, java.net.Socket, java.net.SocketAddress，第 C.2.31 节

C.2.33 **UnresolvedAddressException**

当试图使用不能被分解为真实的网络地址的 SocketAddress 对象时抛出 UnresolvedAddressException（未检查）。

```
    public class  UnresolvedAddressException
       extends IllegalArgumentException
    {
       public  UnresolvedAddressException()
    }
```

参见：java.net.InetSocketAddress, java.net.SocketAddress，第 C.2.32 节

C.2.34 **UnsupportedAddressTypeException**

当试图用 SocketAddress 对象连接套接，该对象表示套接实现不支持的地址类型时，抛出 UnsupportedAddressTypeException（未检查）。

```
    public class UnsupportedAddressTypeException
       extends IllegalArgumentException
    {
       public  UnsupportedAddressTypeException()
    }
```

参见：java.net.InetSocketAddress, java.net.SocketAddress

C.2.35 **WritableByteChannel**

WritableByteChannel 接口定义 *write()*方法，实现在来自 ByteBuffer 的通道上写入数据的操作。

```
public interface WritableByteChannel
   extends Channel
{
   public int write (java.nio.ByteBuffer src)
      throws java.io.IOException;
}
```

参见：第 C.1.4 节，第 C.2.3 节，第 C.2.26 节

C.3 **java.nio.channels.spi 包**

`java.nio.channels.spi` 包包含用来创建可插拔、可选择的通道实现的类。与其他这里列出的包不同，这个包中的类也列出受保护方法。这些类提供可插拔实现可以重新使用的通用方法，但不是所有都打算用于公共消耗。

C.3.1 **AbstractInterruptibleChannel**

`AbstractInterruptibleChannel` 类提供实现子类的终端语义的方法。

```
public abstract class AbstractInterruptibleChannel
   implements java.nio.channels.Channel,
      java.nio.channels.InterruptibleChannel
{
   protected final void begin()
   public final void close()
      throws java.io.IOException
   protected final void end (boolean completed)
      throws java.nio.channels.AsynchronousCloseException
   protected abstract void implCloseChannel()
      throws java.io.IOException;
   public final boolean isOpen()
}
```

C.3.2 **AbstractSelectableChannel**

`AbstractSelectableChannel` 是所有有资格参加就绪选择的通道实现的超类。

```
public abstract class AbstractSelectableChannel
   extends java.nio.channels.SelectableChannel
{
   public final Object blockingLock()
   public final java.nio.channels.SelectableChannel configureBlocking
      (boolean block) throws java.io.IOException
   protected final void implCloseChannel()
      throws java.io.IOException
   protected abstract void implCloseSelectableChannel()
      throws java.io.IOException;
   protected abstract void implConfigureBlocking (boolean block)
      throws java.io.IOException;
   public final boolean isBlocking()
   public final boolean isRegistered()
   public final java.nio.channels.SelectionKey keyFor
      (java.nio.channels.Selector sel)
   public final SelectorProvider provider()
   public final java.nio.channels.SelectionKey register
      (java.nio.channels.Selector sel, int ops, Object att)
```

```
        throws java.nio.channels.ClosedChannelException
}
```

C.3.3 **AbstractSelectionKey**

AbstractSelectionKey 类提供 SelectionKey 实现使用的通用例行程序。

```
public abstract class AbstractSelectionKey
   extends java.nio.channels.SelectionKey
{
   public final void cancel()
   public final boolean isValid()
}
```

C.3.4 **AbstractSelector**

AbstractSelector 类是所有 Selector 实现的超类。

```
public abstract class AbstractSelector
   extends java.nio.channels.Selector
{
   protected final void begin()
   protected final java.util.Set  cancelledKeys()
   public final void close()
      throws java.io.IOException
   protected final void deregister (AbstractSelectionKey key)
   protected final void end()
   protected abstract void implCloseSelector()
      throws java.io.IOException;
   public final boolean isOpen()
   public final SelectorProvider  provider()
   protected abstract java.nio.channels.SelectionKey register(
      AbstractSelectableChannel ch, int ops, Object att);
}
```

C.3.5 **SelectorProvider**

SelectorProvider 类是所有具体通道提供者类的超类。这个类仅被服务器提供者接口设备实例化，从不是直接的。具体子类的全部合格名称都应被列在类加载的类路径中名为 *META-INF/services/java.nio.channels.spi.SelectorProvider* 的文件上。

```
public abstract class SelectorProvider
{
   public abstract java.nio.channels.DatagramChannel
openDatagramChannel()
      throws java.io.IOException;
   public abstract java.nio.channels.Pipe openPipe()
      throws java.io.IOException;
   public abstract AbstractSelector  openSelector()
      throws java.io.IOException;
   public abstract java.nio.channels.ServerSocketChannel
       openServerSocketChannel() throws java.io.IOException;
   public abstract java.nio.channels.SocketChannel openSocketChannel()
      throws java.io.IOException;
   public static SelectorProvider  provider()
}
```

C.4 **java.nio.charset 包**

java.nio.charset 包包含跟字符集处理和转码相关的类。

C.4.1 **CharacterCodingException**

抛出 CharacterCodingException 表示遇到了字符集编码错误。这是在该包中定义的两个特定编码错误异常的父类。低层次的编码器和解码器不抛出这个异常：他们返回 CoderResult 对象表示遇到了哪类错误。在某些情况下，把异常抛出到更高层次的代码要更适当一些。CharsetEncoder.encoder()和 CharsetDecoder.decode()简便方法可能会抛出这个异常。它们是低层次编码方法的简易包装器并且使用 CoderResult.throwException()方法。

```
public class CharacterCodingException
   extends java.io.IOException
{
   public  CharacterCodingException()
}
```

参见：第 C.4.6 节，第 C.4.9 节，第 C.4.10 节

C.4.2 **Charset**

Charset 类封装编码的字符集和相关的编码方案。

```
public abstract class  Charset
   implements Comparable
{
   public final java.util.Set  aliases()
   public static java.util.SortedMap  availableCharsets()
   public boolean  canEncode()
   public final int compareTo (Object ob)
   public abstract boolean contains (Charset cs);
   public final java.nio.CharBuffer decode (java.nio.ByteBuffer bb)
   public String displayName()
   public String displayName (java.util.Locale locale)
   public final java.nio.ByteBuffer  encode (String str)
   public final java.nio.ByteBuffer  encode (java.nio.CharBuffer cb)
   public final boolean equals (Object ob)
   public static Charset forName (String charsetName)
   public final int hashCode()
   public final boolean isRegistered()
   public static boolean isSupported (String charsetName)
   public final String name()
   public abstract CharsetDecoder  newDecoder();
   public abstract CharsetEncoder  newEncoder();
   public final String toString()
}
```

参见：第 C.4.3 节，第 C.4.4 节

C.4.3 **CharsetDecoder**

`CharsetDecoder` 实例把字节的编码序列转化成字符序列。此类的实例有状态。

```
public abstract class CharsetDecoder
{
   public final float averageCharsPerByte()
   public final Charset charset()
   public final java.nio.CharBuffer  decode (java.nio.ByteBuffer in)
       throws CharacterCodingException
   public final CoderResult  decode (java.nio.ByteBuffer in,
      java.nio.CharBuffer out, boolean endOfInput)
   public Charset  detectedCharset()
   public final CoderResult flush (java.nio.CharBuffer out)
   public boolean isAutoDetecting()
   public boolean  isCharsetDetected()
   public CodingErrorAction  malformedInputAction()
   public final float  maxCharsPerByte()
   public final CharsetDecoder  onMalformedInput (
      CodingErrorAction newAction)
   public final CharsetDecoder  onUnmappableCharacter (
      CodingErrorAction newAction)
   public final CharsetDecoder  replaceWith (String newReplacement)
   public final String replacement()
   public final CharsetDecoder  reset()
   public CodingErrorAction  unmappableCharacterAction()
}
```

参见：第 C.4.2 节，第 C.4.4 节

C.4.4 **CharsetEncoder**

`CharsetEncoder` 实例把字符序列转化成字节的编码序列。此类的实例有状态。

```
public abstract class CharsetEncoder
{
   public final float averageBytesPerChar()
   public boolean  canEncode (char c)
   public boolean  canEncode (CharSequence cs)
   public final Charset charset()
   public final java.nio.ByteBuffer  encode (java.nio.CharBuffer in)
       throws CharacterCodingException
   public final CoderResult   encode (java.nio.CharBuffer in,
      java.nio.ByteBuffer out, boolean endOfInput)
   public final CoderResult  flush (java.nio.ByteBuffer out)
   public boolean  isLegalReplacement (byte [] repl)
   public CodingErrorAction  malformedInputAction()
   public final float maxBytesPerChar()
   public final CharsetEncoder  onMalformedInput (
      CodingErrorAction newAction)
   public final CharsetEncoder  onUnmappableCharacter (
      CodingErrorAction newAction)
   public final CharsetEncoder  replaceWith (byte [] newReplacement)
   public final byte [] replacement()
   public final CharsetEncoder  reset()
   public CodingErrorAction  unmappableCharacterAction()
}
```

参见：第 C.4.2 节，第 C.4.3 节

C.4.5 **CoderMalfunctionError**

当 *CharsetEncoder.encode()*或 *CharsetDecoder.decode()*方法从低层次的 *encodeLoop()*或 *decodeLoop()*方法中捕捉到意外的异常时抛出 `CoderMalfunctionError`。

```
public class  CoderMalfunctionError
   extends Error
{
   public  CoderMalfunctionError (Exception cause)
}
```

参见：第 C.4.3 节，第 C.4.4 节

C.4.6 **CoderResult**

`CoderResult` 对象是由 *CharsetDecoder.decode()*和 *CharsetEncoder.encode()*返回，表示编码运行的结果。

```
public class  CoderResult
{
   public static final CoderResult  OVERFLOW
   public static final CoderResult  UNDERFLOW
   public boolean isError()
   public boolean isMalformed()
   public boolean isOverflow()
   public boolean isUnderflow()
   public boolean isUnmappable()
   public int length()
   public static CoderResult malformedForLength (int length)
   public void  throwException()
      throws CharacterCodingException
   public String toString()
   public static CoderResult  unmappableForLength (int length)
}
```

参见：第 C.4.1 节，第 C.4.3 节，第 C.4.4 节

C.4.7 **CodingErrorAction**

`CodingErrorAction` 类是安全类型列举。命名的实例被导入到 `CharsetDecoder` 和 `CharsetEncoder` 对象中，表示当遇到编码错误时应该采取哪种行动。

```
public class  CodingErrorAction
{
   public static final CodingErrorAction IGNORE
   public static final CodingErrorAction REPLACE
   public static final CodingErrorAction REPORT
   public String toString()
}
```

参见：第 C.4.3 节，第 C.4.4 节，第 C.4.6 节

C.4.8 **IllegalCharsetNameException**

当 Charset 名称与提供的字符集命名规则不兼容时抛出 IllegalCharsetNameException（未检查）。字符集名称必须由 ASCII 字母（大写或小写），阿拉伯数字，连字符，冒号，下划线和句点组成，并且首字符必须为字母或阿拉伯数字。

```
public class IllegalCharsetNameException
   extends IllegalArgumentException
{
   public  IllegalCharsetNameException (String charsetName)
   public String getCharsetName()
}
```

参见：第 C.4.2 节

C.4.9 **MalformedInputException**

MalformedInputException（IOException 的子类）被抛出表示在编码运行期间探测到了有缺陷的输入。需要时 CoderResult 对象提供产生这个异常的简便方法。

```
public class MalformedInputException
   extends CharacterCodingException
{
   public MalformedInputException (int inputLength)
   public int  getInputLength()
   public String getMessage()
}
```

参见：第 C.4.6 节，第 C.4.10 节

C.4.10 **UnmappableCharacterException**

抛出 UnmappableCharacterException（IOException 的子类）表示编码器或解码器不能从另外的有效输入序列中映射一个或多个字符。CoderResult 对象提供产生这个异常的简便方法。

```
public class UnmappableCharacterException
   extends CharacterCodingException
{
   public  UnmappableCharacterException (int inputLength)
   public int  getInputLength()
   public String getMessage()
}
```

参见：第 C.4.6 节，第 C.4.9 节

C.4.11 **UnsupportedCharsetException**

当前的 JVM 环境不支持必要 Charset 时抛出 UnsupportedCharsetException（未检查）。

```
public class UnsupportedCharsetException
   extends IllegalArgumentException
{
   public UnsupportedCharsetException (String charsetName)
   public String getCharsetName()
}
```

参见：第 C.4.2 节

C.5 **java.nio.charset.spi 包**

`java.nio.charset.spi` 包包含字符集服务器提供者接口机制使用的一个独立的类。

C.5.1 **CharsetProvider**

`CharsetProvider` 促进 `Charset` 实现到运行 JVM 中的安装。具体子类的全部合格名称应该在类加载器的类路径中名为 *META-INF/services/java.nio.charset.spi.CharsetProvider* 的文件中列出，通过服务器提供者接口机制激活他们。

```
public abstract class CharsetProvider
{
   public abstract java.nio.charset.Charset charsetForName (
      String charsetName);
   public abstract java.util.Iterator  charsets();
}
```

参见：第 C.4.2 节

C.6 **java.util.regex 包**

`java.util.regex` 包包含用于正规表达式处理的类。

C.6.1 **Matcher**

`Matcher` 对象是状态匹配引擎，检查输入字符序列用来探测正规表达式匹配并提供有关成功匹配的信息。

```
public final class Matcher
{
   public Matcher  appendReplacement (StringBuffer sb, String
replacement)
   public StringBuffer appendTail (StringBuffer sb)
   public int  end()
   public int  end (int group)
   public boolean  find()
   public boolean  find (int start)
   public String group()
   public String group (int group)
   public int  groupCount()
   public boolean  lookingAt()
   public boolean  matches()
```

```
    public Pattern  pattern()
    public String replaceAll (String replacement)
    public String replaceFirst (String replacement)
    public Matcher  reset()
    public Matcher  reset (CharSequence input)
    public int start()
    public int start (int group)
}
```

参见：java.lang.CharSequence, java.lang.String， 第 C.6.2 节

C.6.2 **Pattern**

Pattern 类封装编译的正规表达式。

```
public final class Pattern
    implements java.io.Serializable
{
    public static final int CANON_EQ
    public static final int CASE_INSENSITIVE
    public static final int COMMENTS
    public static final int DOTALLf
    public static final int MULTILINE
    public static final int UNICODE_CASE
    public static final int UNIX_LINES
    public static Pattern compile (String regex)
    public static Pattern compile (String regex, int flags)
    public int  flags()
    public Matcher  matcher (CharSequence input)
    public static boolean matches (String regex, CharSequence input)
    public String pattern()
    public String [] split (CharSequence input)
    public String [] split (CharSequence input, int limit)
}
```

参见：java.lang.CharSequence, java.lang.String, 第 C.6.1 节

C.6.3 **PatternSyntaxException**

当提供的正规表达式字符串包含句法错误时通过 *Pattern.compile()*（或在 Pattern 或 String 上采用正规表达式参数的任意简便方法）抛出 PatternSyntaxException.

```
public class PatternSyntaxException
    extends IllegalArgumentException
{
    public  PatternSyntaxException (String desc, String regex, int index)
    public String getDescription()
    public int  getIndex()
    public String getMessage()
    public String getPattern()
}
```

参见：第 C.6.2 节